

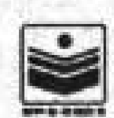
中国计算机学会教育专业委员会
全国高等学校计算机教育研究会
推荐出版

高等学校规划教材

计算机算法设计与分析（第2版）

王晓东 编著

计算机学科教学计划2001



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn







ISBN 7-121-00001-6

9 787121 000010 >ISBN 7-121-00001-6

定价：25.50 元

高等学校规划教材

计算机算法设计与分析

（第 2 版）

王晓东　编著

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书为大学计算机专业核心课程算法设计与分析教材。全书以算法设计策略为知识单元，系统介绍算法设计方法与分析技巧。主要内容包括：算法概述、递归与分治策略、动态规划、贪心算法、回溯法、分支限界法、概率算法、线性规划与网络流、NP完全性理论与近似算法等。书中既涉及经典与实用算法及实例分析，又包括算法领域热点追踪。

为突出教材的可读性和可用性，章首增加了学习要点提示，章末配有难易适度的习题，并免费提供电子课件和其他教学参考资料（包括习题解题思路提示和上机实验安排等）。任课教师可按前言中所提供的方式索取。

本书适合于作为大学计算机科学与技术及相关专业本科生和研究生教材，也适合广大工程技术人员学习参考。

图书在版编目（CIP）数据

计算机算法设计与分析/王晓东编著．—2版．—北京：电子工业出版社，2004.7

高等学校规划教材

ISBN 7-121-00001-6

Ⅰ．计… Ⅱ．王… Ⅲ．①电子计算机-算法设计-高等学校-教材②电子计算机-算法分析-高等学校-教材 Ⅳ．TP301.6

中国版本图书馆CIP数据核字（2004）第056034号

策划编辑：童占梅
责任编辑：童占梅
印　　刷：北京大中印刷厂
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路173信箱　邮编100036
经　　销：各地新华书店
开　　本：787×1092　1/16　印张：21.25　字数：539千字
印　　次：2004年7月第1次印刷
印　　数：10100册　　定　　价：25.50元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

新版说明

由中国计算机学会教育专业委员会和全国高等学校计算机教育研究会(简称"两会")组织和推荐,自1996年起电子工业出版社出版了基于CC1991教程的15本系列教材。该系列教材受到高校师生和读者的普遍欢迎和肯定,其中有11本入选1996—2000年全国工科电子类专业规划教材。

几年过去了,计算机学科又有了很大发展。IEEE-CS/ACM联合计算教程专题组,组织世界各国150多位专家,历时3年多,在美、欧、亚召开了一系列会议,在CC1991的基础上,发布了"Computing Curricula 2001 - Computer Science Final Report"(简称CC2001)。专家们认为:随着计算机学科技术的迅速发展,使得现有的任何一所学校的计算机专业都很难再像CC1991所提到的那样,能够覆盖计算机学科的所有知识领域。所以,需按市场需求将计算机学科划分为4个主要分支:计算机科学、计算机工程、软件工程和信息系统。其中计算机科学是各分支的基础,CC2001正是基于计算机科学制订的。我国"两会"追踪CC2001,经过3年多的工作,最后以中国计算机科学与技术教程2002研究组的名义推出了"China Computing Curricula 2002"(简称CCC2002)。CC2001与CC1991比较有以下几个方面的变化:

(1)将CC1991确定的11个主领域扩展为14个主领域:离散结构、编程基础、算法与复杂性、计算机组织与体系结构、操作系统、网络计算、编程语言、人-机交互、图形学与可视化计算、智能系统、信息管理、职业与社会问题、软件工程、数值计算。对各主领域的名称、核心内容及选学内容都进行了调整和扩充。

(2)提出了课程的组织结构和实现策略。课程分为3类:入门(基础)课程、核心(必修)课程和附加(选修)课程。入门课程可按编程、算法和硬件优先等多种方式组织,使学生能够接触到计算机系统的设计、构造和应用,为学生提供实用性的技能训练,同时还应提高学生的兴趣和智慧;核心课程的组织可按传统、压缩、系统或网络方法进行,特别强调贯彻CC1991提出的3个过程、12个重复概念、职业与社会的关系等方法论思想;此外,还应设置一些介绍热门或前沿技术的附加课程。

(3)更加强调学生的专业实践,要求把专业实践放在重要位置,并贯穿于教学的全过程。

这次对系列教材的全面修版,力求反映计算机学科发展的最新成就,并力争符合CC2001和CCC2002所提出的要求及高校课程和教学改革的需要。这套教材的对象为本科生、研究生和大专生(通过删减使用)。信息技术领域的从业人员也可使用。

为了保证编审和出版质量,编委会进行了调整,电子工业出版社成立了编辑出版小组。在原教材工作的基础上,编委会对教材大纲逐一进行了认真讨论和评审,其中一些关键性和难度较大的教材还进行了多次讨论和修改。

限于水平和经验,教材中还会存在缺点和不足,希望读者提出中肯的批评和建议。读者可以通过电子工业出版社华信教育资源网站 http://www.hxedu.com.cn 反馈信息并发表意见,我们在此表示衷心的感谢!

教材编委会

前　言

计算机的普及极大地改变了人们的生活。目前，各行业、各领域都广泛采用了计算机信息技术，并由此产生出开发各种应用软件的需求。为了以最少的成本、最快的速度、最好的质量开发出适合各种应用需求的软件，必须遵循软件工程的原则。设计一个高效的程序不仅需要编程小技巧，更需要合理的数据组织和清晰高效的算法，这正是计算机科学领域数据结构与算法设计所研究的主要内容。一些著名的计算机科学家在有关计算机科学教育的论述中认为，计算机科学是一种创造性思维活动，其教育必须面向设计。计算机算法设计与分析正是一门面向设计，且处于计算机学科核心地位的教育课程。通过对计算机算法系统的学习与研究，掌握算法设计的主要方法，培养对算法的计算复杂性正确分析的能力，为独立设计算法和对算法进行复杂性分析奠定坚实的理论基础，对每一位从事计算机系统结构、系统软件和应用软件研究与开发的科技工作者都是非常重要和必不可少的。为了适应 21 世纪我国培养计算机各类人才的需要，本课程结合我国高等学校教育工作的现状，追踪国际计算机科学技术的发展水平，更新了教学内容和教学方法，以算法设计策略为知识单元，系统地介绍计算机算法的设计方法与分析技巧，以期为计算机专业的学生提供一个广泛扎实的计算机算法知识基础。

本书第 2 版修正了第 1 版中已发现的一些错误，并将第 1 版的第 8 章和第 9 章合并为第 9 章，增加了第 8 章线性规划与网络流算法的有关内容。

全书共分 9 章，第 1 章介绍算法的基本概念，并对算法的计算复杂性和算法的描述作了简要阐述。然后围绕算法设计常用的基本设计策略组织了第 2 章至第 9 章的内容。

第 2 章介绍递归与分治策略，它是设计有效算法最常用的策略，也是必须掌握的方法。

第 3 章是动态规划算法，以具体实例详述动态规划算法的设计思想、适用性以及算法的设计要点。

第 4 章介绍贪心算法，它也是一种重要的算法设计策略，它与动态规划算法的设计思想有一定的联系，但其效率更高。按贪心算法设计出的许多算法能导致最优解。其中有许多典型问题和典型算法可供学习和使用。

第 5 章和第 6 章分别介绍回溯法和分支限界法。这两章所介绍的算法适合于处理难解问题。其解题思想各具特色，值得学习和掌握。

第 7 章介绍概率算法，对许多难解问题提供了高效的解决途径，是有很高实用价值的算法设计策略。

第 8 章介绍实用性很强的线性规划与网络流算法。许多实际应用问题可以转化为线性规划和网络流问题，并可用第 8 章中的算法有效求解。

第 9 章首先介绍计算模型、确定性和非确定性图灵机，然后进一步深入介绍 NP 完全性理论和 NP 难问题的近似算法，这是当前计算机算法领域的热点研究课题，具有很高的实用价值。

在本书各章的论述中，首先介绍一种算法设计策略的基本思想，然后从解决计算机科学和应用中的实际问题入手，由简到繁地描述几个经典的精巧算法。同时对每个算法所需的时间和空间进行分析，使读者既能学到一些常用的精巧算法，又能通过对算法设计策略的反复应用，牢固掌握这些算法设计的基本策略，以期收到融会贯通之效。在为各种算法设计策略选择用于展示其设计思想与技巧的具体应用问题时，本书有意重复选择某些经典问题，使读者能深

刻地体会到一个问题可以用多种设计策略求解。同时通过对解同一问题的不同算法的比较，使读者更容易体会到每一种具体算法的设计要点。随着本书内容的逐步展开，读者也将进一步感受到综合应用多种设计策略可以更有效地解决问题。

本书采用面向对象的C++语言作为算法描述手段，在保持C++优点的同时，尽量使算法描述简明、清晰。

为便于学习，我们在章首增加了学习要点提示，在章末配有难易适度的习题。为便于教学，本教材将免费提供电子课件和其他教学参考资料(包括习题解题思路提示和上机实验安排等)。请任课教师登录电子工业出版社华信教育资源网 http://www.hxedu.com.cn 或直接联系教材服务部 010-68152204 索取。

在本书编写过程中，得到了全国高等学校计算机专业教学指导委员会的关心和支持。福州大学"211工程"计算机与信息工程重点学科实验室为本书的写作提供了优良的设备和工作环境。电子工业出版社负责本书编辑出版工作的全体同仁为本书的出版付出了大量辛勤的劳动，他们认真细致、一丝不苟的工作精神保证了本书的出版质量。傅清祥教授在百忙之中认真审阅了全书，提出了许多宝贵的改进意见。在此，谨向每一位曾经关心和支持本书编写工作的各方面人士表示衷心的谢意!

由于作者的知识和写作水平有限，书稿虽几经修改，仍难免有缺点和错误。热忱欢迎同行专家和读者批评指正，使本书在使用中不断得到改进，日臻完善。作者 E-mail: wangxd@fzu.edu.cn。

作　者

目　　录

第1章　算法概述

学习要点

- 理解算法的概念
- 理解什么是程序，程序与算法的区别和内在联系
- 掌握求解问题的基本步骤
- 掌握算法在最坏情况、最好情况和平均情况下的计算复杂性概念
- 掌握算法复杂性的渐近性态的数学表述
- 掌握用C++语言描述算法的方法

1.1　算法与程序

对于计算机科学来说，算法(Algorithm)的概念是至关重要的。例如在一个大型软件系统的开发中，设计出有效的算法将起决定性的作用。通俗地讲，算法是指解决问题的一种方法或一个过程。更严格地讲，算法是由若干条指令组成的有穷序列，且满足下述几条性质：

(1) 输入：有零个或多个由外部提供的量作为算法的输入。

(2) 输出：算法产生至少一个量作为输出。

(3) 确定性：组成算法的每条指令是清晰的，无歧义的。

(4) 有限性：算法中每条指令的执行次数是有限的，执行每条指令的时间也是有限的。

程序(Program)与算法不同。程序是算法用某种程序设计语言的具体实现。程序可以不满足算法的性质(4)。例如操作系统，它是一个在无限循环中执行的程序，因而不是一个算法。然而我们可把操作系统的各种任务看成是一些单独的问题，每一个问题由操作系统中的一个子程序通过特定的算法来实现。该子程序得到输出结果后便终止。

描述算法可以有多种方式，如自然语言方式、表格方式等。在本书中，我们采用C++语言来描述算法。C++语言的优点是类型丰富、语句精练，具有面向过程和面向对象的双重特点。用C++来描述算法可使整个算法结构紧凑，可读性强。在本书中，有时为了更好地阐明算法的思路，我们还采用C++与自然语言相结合的方式来描述算法

1.2　算法复杂性分析

一个算法的复杂性的高低体现在运行该算法所需要的计算机资源的多少上，所需资源越多，我们就说该算法的复杂性越高；反之，所需资源越少，我们就说该算法的复杂性越低。最重要的计算机资源是时间和空间(即存储器)资源。因此，算法的复杂性有时间复杂性和空间复杂性之分。

不言而喻，对于任意给定的问题，设计出复杂性尽可能低的算法是我们在设计算法时追求的一个重要目标。另一方面，当给定的问题已有多种算法时，选择其中复杂性最低者，是我们在

选用算法时遵循的一个重要准则 因此,算法的复杂性分析对算法的设计或选用有着重要的指导意义和实用价值。

算法的复杂性是算法运行所需要的计算机资源的量,需要时间资源的量称为时间复杂性,需要的空间资源的量称为空间复杂性。这个量应该集中反映算法的效率,并从运行该算法的实际计算机中抽象出来。换句话说,这个量应该是只依赖于要解的问题的规模、算法的输入和算法本身的函数。如果分别用 N, I 和 A 表示算法要解的问题的规模、算法的输入和算法本身,而且用 C 表示复杂性,那么,应该有 $C = F(N, I, A)$,其中 $F(N, I, A)$ 是一个由 N, I 和 A 确定的三元函数。如果把时间复杂性和空间复杂性分开,并分别用 T 和 S 来表示,应该有:$T = T(N, I, A)$ 和 $S = S(N, I, A)$。通常,我们让 A 隐含在复杂性函数名当中,因而将 T 和 S 分别简写为 $T = T(N, I)$ 和 $S = S(N, I)$。

由于时间复杂性与空间复杂性概念类同,计量方法相似,且空间复杂性分析相对简单些,所以本书将主要讨论时间复杂性。现在的问题是如何将复杂性函数具体化,即对于给定的 N、I 和 A,如何导出 $T(N, I)$ 和 $S(N, I)$ 的数学表达式,来给出计算 $T(N, I)$ 和 $S(N, I)$ 的法则。下面以 $T(N, I)$ 为例,将复杂性函数具体化。

根据 $T(N, I)$ 的概念,它应该是算法在一台抽象的计算机上运行所需要的时间。设此抽象的计算机所提供的元运算有 k 种,它们分别记为 $O_1, O_2, \cdots, O_k$。又设每执行一次这些元运算所需要时间分别为 $t_1, t_2, \cdots, t_k$。对于给定的算法 A,设经统计,用到元运算 O_i 的次数为 e_i, $i = 1, 2, \cdots, k$。很清楚,对于每一个 $i, 1 \leq i \leq k$, e_i 是 N 和 I 的函数,即 $e_i = e_i(N, I)$。那么有

$$T(N, I) = \sum_{i=1}^{k} t_i e_i(N, I)$$

其中,$t_i(i = 1, 2, \cdots, k)$ 是与 N 和 I 无关的常数。

显然,我们不可能对规模 N 的每一种合法的输入 I 都去统计 $e_i(N, I)$, $i = 1, 2, \cdots, k$。因此 $T(N, I)$ 的表达式还要进一步简化,或者说,我们只能在规模为 N 的某些或某类有代表性的合法输入中统计相应的 e_i, $i = 1, 2, \cdots, k$,评价其时间复杂性。

本书只考虑三种情况下的时间复杂性,即最坏情况、最好情况和平均情况下的时间复杂性,并分别记为 $T_{\max}(N)$、$T_{\min}(N)$ 和 $T_{\text{avg}}(N)$。在数学上有

$$T_{\max}(N) = \max_{I \in D_N} T(N, I) = \max_{I \in D_N} \sum_{i=1}^{k} t_i e_i(N, I) = \sum_{i=1}^{k} t_i e_i(N, I^*) = T(N, I^*)$$

$$T_{\min}(N) = \min_{I \in D_N} T(N, I) = \min_{I \in D_N} \sum_{i=1}^{k} t_i e_i(N, I) = \sum_{i=1}^{k} t_i e_i(N, \tilde{I}) = T(N, \tilde{I})$$

$$T_{\text{avg}}(N) = \sum_{I \in D_N} P(I) T(N, I) = \sum_{I \in D_N} P(I) \sum_{i=1}^{k} t_i e_i(N, I)$$

式中,D_N 是规模为 N 的合法输入的集合;I^* 是 D_N 中一个使 $T(N, I^*)$ 达到 $T_{\max}(N)$ 的合法输入;$\tilde{I}$ 是 D_N 中一个使 $T(N, \tilde{I})$ 达到 $T_{\min}(N)$ 的合法输入;而 $P(I)$ 是在算法的应用中出现输入 I 的概率。

以上三种情况下的时间复杂性从不同角度来反映算法的效率,各有其局限性,也各有各的用处。实践表明可操作性最好且最有实际价值的是最坏情况下的时间复杂性。本书对算法的时间复杂性分析的重点将放在这种情形上。

随着经济的发展、社会的进步和科学研究的深入,要求用计算机解决的问题越来越复杂,规模越来越大。对求解这类问题的算法作复杂性分析具有特别重要的意义,因而要特别关注。

在此，我们引入复杂性渐近性态的概念。

设 $T(N)$ 是前面所定义的关于算法 A 的复杂性函数。一般说来，当 N 单调增加且趋于 ∞ 时，$T(N)$ 也将单调增加趋于 ∞。对于 $T(N)$，如果存在 $\tilde{T}(N)$，使得当 $N \to \infty$ 时有 $(T(N) - \tilde{T}(N))/T(N) \to 0$，那么，我们就说 $\tilde{T}(N)$ 是 $T(N)$ 当 $N \to \infty$ 时的渐近性态，或叫 $\tilde{T}(N)$ 为算法 A 当 $N \to \infty$ 的渐近复杂性而与 $T(N)$ 相区别。因为在数学上，$\tilde{T}(N)$ 是 $T(N)$ 当 $N \to \infty$ 时的渐近表达式，直观上，$\tilde{T}(N)$ 是 $T(N)$ 中略去低阶项所留下的主项，所以它无疑比 $T(N)$ 来得简单。比如当 $T(N) = 3N^2 + 4N\log N + 7$ 时，$\tilde{T}(N)$ 的一个答案是 $3N^2$，因为这时有

$$(T(N) - \tilde{T}(N))/T(N) = \frac{4N\log N + 7}{3N^2 + 4N\log N + 7} \to 0(\text{当 } N \to \infty \text{ 时})$$

显然，$3N^2$ 比 $3N^2 + 4N\log N + 7$ 简单得多。

由于当 $N \to \infty$ 时，$T(N)$ 渐近于 $\tilde{T}(N)$，我们有理由用 $\tilde{T}(N)$ 来替代 $T(N)$ 作为算法 A 在 $N \to \infty$ 时的复杂性的度量。而且由于 $\tilde{T}(N)$ 明显地比 $T(N)$ 简单，这种替代是对复杂性分析的一种简化。进一步考虑到分析算法的复杂性的目的在于比较求解同一问题的两个不同算法的效率，而当要比较的两个算法的渐近复杂性的阶不相同时，只要能确定出各自的阶，就可以判定哪一个算法的效率高。换句话说，这时的渐近复杂性分析只要关心 $\tilde{T}(N)$ 的阶就够了，不必关心包含在 $\tilde{T}(N)$ 中的常数因子。所以，我们常常又对 $\tilde{T}(N)$ 的分析进一步简化，即假设算法中用到的所有不同的元运算各执行一次所需要的时间都是一个单位时间。

综上所述，我们已经给出了简化算法复杂性分析的方法和步骤，即只要考察当问题的规模充分大时，算法复杂性在渐近意义下的阶。本书的算法分析都将这么做。为此引入以下渐近意义下的记号 O, Ω, θ 和 o。

以下设 $f(N)$ 和 $g(N)$ 是定义在正数集上的正函数。

如果存在正的常数 C 和自然数 N_0，使得当 $N \geqslant N_0$ 时有 $f(N) \leqslant Cg(N)$，则称函数 $f(N)$ 当 N 充分大时上有界，且 $g(N)$ 是它的一个上界，记为 $f(N) = O(g(N))$。这时我们还说 $f(N)$ 的阶不高于 $g(N)$ 的阶。

举几个例子：

(1) 因为对所有的 $N \geqslant 1$ 有 $3N \leqslant 4N$，我们有 $3N = O(N)$；

(2) 因为当 $N \geqslant 1$ 时有 $N + 1024 \leqslant 1025N$，我们有 $N + 1024 = O(N)$；

(3) 因为当 $N \geqslant 10$ 时有 $2N^2 + 11N - 10 \leqslant 3N^2$，我们有 $2N^2 + 11N - 10 = O(N^2)$；

(4) 因为对所有 $N \geqslant 1$ 有 $N^2 \leqslant N^3$，我们有 $N^2 = O(N^3)$；

(5) 作为一个反例 $N^3 \neq O(N^2)$。因为若不然，则存在正的常数 C 和自然数 N_0，使得当 $N \geqslant N_0$ 有 $N^3 \leqslant CN^2$，即 $N \leqslant C$。显然，当取 $N = \max\{N_0, \lfloor C \rfloor + 1\}$ 时这个不等式不成立，所以 $N^3 \neq O(N^2)$。

按照符号 O 的定义，容易证明它有如下运算性质：

(1) $O(f) + O(g) = O(\max(f, g))$。

(2) $O(f) + O(g) = O(f + g)$。

(3) $O(f)O(g) = O(fg)$。

(4) 如果 $g(N) = O(f(N))$,则 $O(f) + O(g) = O(f)$。

(5) $O(Cf(N)) = O(f(N))$,其中 C 是一个正的常数。

(6) $f = O(f)$。

性质(1) 的证明:设 $F(N) = O(f)$。根据符号 O 的定义,存在正常数 C_1 和自然数 N_1,使得对所有的 $N \geqslant N_1$,有 $F(N) \leqslant C_1 f(N)$。

类似地,设 $G(N) = O(g)$,则存在正的常数 C_2 和自然数 N_2,使得对所有的 $N \geqslant N_2$ 有 $G(N) \leqslant C_2 g(N)$。

令 $C_3 = \max\{C_1, C_2\}, N_3 = \max\{N_1, N_2\}, h(N) = \max\{f, g\}$。则对所有的 $N \geqslant N_3$,有

$$F(N) \leqslant C_1 f(N) \leqslant C_1 h(N) \leqslant C_3 h(N)$$

类似地,有

$$G(N) \leqslant C_2 f(N) \leqslant C_2 h(N) \leqslant C_3 h(N)$$

因而

$$\begin{aligned} O(f) + O(g) &= F(N) + G(N) \\ &\leqslant C_3 h(N) + C_3 h(N) \\ &= 2C_3 h(N) \\ &= O(h) \\ &= O(\max(f, g)) \end{aligned}$$

其余性质的证明类似,留给读者作为练习。

应该指出,根据符号 O 的定义,用它评估算法的复杂性,得到的只是当规模充分大时的一个上界。这个上界的阶越低则评估就越精确,结果就越有价值。

关于符号 Ω,文献里有两种不同的定义。本书只采用其中的一种,定义如下:如果存在正的常数 C 和自然数 N_0,使得当 $N \geqslant N_0$ 时有 $f(N) \geqslant Cg(N)$,则称函数 $f(N)$ 当 N 充分大时下有界;且 $g(N)$ 是它的一个下界,记为 $f(N) = \Omega(g(N))$。这时我们还说 $f(N)$ 的阶不低于 $g(N)$ 的阶。Ω 的这个定义的优点是与 O 的定义对称,缺点是当 $f(N)$ 对自然数的不同无穷子集有不同的表达式,且有不同的阶时,不能很好地刻画出 $f(N)$ 的下界。比如当

$$f(N) = \begin{cases} 100 & N \text{ 为正偶数} \\ 6N^2 & N \text{ 为正奇数} \end{cases}$$

时,按上述定义,得到 $f(N) = \Omega(1)$,这是一个平凡的下界,对算法分析没有什么价值。然而,考虑到上述定义有与符号 O 定义的对称性,本书还是选用它。

同样地,用 Ω 评估算法的复杂性,得到的只是该复杂性的一个下界。这个下界的阶越高,则评估就越精确,结果就越有价值。再则,这里的 Ω 只对问题的一个算法而言。如果它是对一个问题的所有算法或某类算法而言,即对于一个问题和任意给定的充分大的规模 N,下界在该问题的所有算法或某类算法的复杂性中取,那么它将更有意义。这时得到的相应下界,我们称之为问题的下界或某类算法的下界。它常常与符号 O 配合以证明某问题的一个特定算法是该问题的最优算法或该问题的某算法类中的最优算法。

我们定义 $f(N) = \theta(g,(N))$ 当且仅当 $f(N) = O(g(N))$ 且 $f(N) = \Omega(g(N))$。这时,我们说 $f(N)$ 与 $g(N)$ 同阶。

最后,我们来看符号 o 的定义。如果对于任意给定的 $\varepsilon > 0$,都存在正整数 N_0,使得当 $N \geqslant N_0$ 时有 $f(N)/g(N) < \varepsilon$,则称函数 $f(N)$ 当 N 充分大时的阶比 $g(N)$ 低,记为 $f(N) = o(g(N))$。

例如：$4N\log N + 7 = o(3N^2 + 4N\log N + 7)$。

习题 1

1-1　求下列函数的渐近表达式：

$3n^2 + 10n$；$n^2/10 + 2^n$；$21 + 1/n$；$\log n^3$；$10\log 3^n$。

1-2　试论 $O(1)$ 和 $O(2)$ 的区别

1-3　画出下列表达式的函数图像，并说明各表达式当 n 在什么范围内取值时效率最高，

$$4n^2、\log n，3^n，20n，2，n^{2/3}$$

1-4　按照渐近阶从低到高的顺序排列以下表达式：$4n^2$，$\log n$，3^n，$20n$，2，$n^{2/3}$。又 $n!$ 应该排在哪一位？

1-5　(1) 假设某算法在输入规模为 n 时的计算时间为 $T(n) = 3 \times 2^n$。在某台计算机上实现并完成该算法的时间为 t 秒。现有另一台计算机，其运行速度为第一台的 64 倍，那么在这台新机器上用同一算法在 t 秒内能解输入规模为多大的问题？

(2) 若上述算法的计算时间改进为 $T(n) = n^2$，其余条件不变，则在新机器上用 t 秒时间能解输入规模为多大的问题？

(3) 若上述算法的计算时间进一步改进为 $T(n) = 8$，其余条件不变，那么我们在新机器上用 t 秒时间能解输入规模为多大的问题？

1-6　硬件厂商 XYZ 公司宣称他们最新研制的微处理器运行速度为其竞争对手 ABC 公司同类产品的 100 倍。对于计算复杂性分别为 n，n^2，n^3 和 $n!$ 的各算法，若用 ABC 公司的计算机在 1 小时内能解输入规模为 n 的问题，那么用 XYZ 公司的计算机在 1 小时内分别能解输入规模为多大的问题？

1-7　对于下列各组函数 $f(n)$ 和 $g(n)$，确定 $f(n) = O(g(n))$ 或 $f(n) = \Omega(g(n))$ 或 $f(n) = \theta(g(n))$，并简述理由。

(1) $f(n) = \log n^2$；$g(n) = \log n + 5$

(2) $f(n) = \log n^2$；$g(n) = \sqrt{n}$

(3) $f(n) = n$；$g(n) = \log^2 n$

(4) $f(n) = n\log n + n$；$g(n) = \log n$

(5) $f(n) = 10$；$g(n) = \log 10$

(6) $f(n) = \log^2 n$；$g(n) = \log n$

(7) $f(n) = 2^n$；$g(n) = 100n^2$

(8) $f(n) = 2^n$；$g(n) = 3^n$

1-8　证明：$n! = o(n^n)$。

1-9　下面的算法段用于确定 n 的初始值。试分析该算法段所需计算时间的上界和下界。

```
while(n > 1)
  if(odd(n))
    n = 3 * n + 1;
  else
    n = n/2;
```

1-10　证明：如果一个算法在平均情况下的计算时间复杂性为 $\theta(f(n))$，则该算法在最坏情况下所需的计算时间为 $\Omega(f(n))$。

第2章　递归与分治策略

学习要点

- 理解递归的概念
- 掌握设计有效算法的分治策略
- 通过下面的范例学习分治策略设计技巧：

(1) 二分搜索技术

(2) 大整数乘法大整数乘法

(3) Strassen 矩阵乘法

(4) 棋盘覆盖

(5) 合并排序和快速排序

(6) 线性时间选择

(7) 最接近点对问题

(8) 循环赛日程表

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，解题所需的计算时间往往也越少，从而也较容易处理。例如，对于 n 个元素的排序问题，当 $n = 1$ 时，不需任何计算。$n = 2$ 时，只要作一次比较即可排好序；$n = 3$ 时只要作两次比较即可……而当 n 较大时，问题就不那么容易处理了。要想直接解决一个较大的问题，有时是相当困难的。分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。如果原问题可分割成 k 个子问题，$1 < k \leqslant n$，且这些子问题都可解，并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易求出其解。这样，就自然导致递归算法的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

2.1　递归的概念

一个直接或间接地调用自身的算法称为递归算法。一个使用函数自身给出定义的函数称为递归函数。在计算机算法设计与分析中，使用递归技术往往使函数的定义和算法的描述简捷且易于理解。有些数据结构如二叉树等，由于其本身固有的递归特性，特别适合用递归的形式来描述。还有一些问题，虽然其本身并没有明显的递归结构，但用递归技术来求解使设计出的算法简洁易懂且易于分析。下面我们来看几个实例。

【例2-1】　阶乘函数

阶乘函数可递归地定义为

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

阶乘函数的自变量 n 的定义域是非负整数。递归式的第一式给出了这个函数的一个初始值，是非递归定义的。每个递归函数都必须有非递归定义的初始值，否则，递归函数就无法计算。递归式的第二式是用较小自变量的函数值来表示较大自变量的函数值的方式来定义 n 的阶乘。定义式的左右两边都引用了阶乘记号，是一个递归定义式，可递归地计算如下：

```
int Factorial(int n)
{
  if (n = = 0) return 1;
  return n * Factorial(n - 1);
}
```

【例2-2】 Fibonacci 数列

无穷数列 1,1,2,3,5,8,13,21,34,55,…，称为 Fibonacci 数列。它可以递归地定义为

$$F(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

这是一个递归关系式，它说明当 n 大于 1 时，这个数列的第 n 项的值是它前面两项之和。它用两个较小的自变量的函数值来定义一个较大自变量的函数值，所以需要两个初始值 $F(0)$ 和 $F(1)$。

第 n 个 Fibonacci 数可递归地计算如下：

```
int Fibonacci(int n)
{
  if (n < = 1) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

上述两例中的函数也可用如下非递归方式定义：

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

并非一切递归函数都能用非递归方式定义。为了对递归函数的复杂性有更多的了解，我们再介绍一个双递归函数——Ackerman 函数。当一个函数及它的一个变量是由函数自身定义时，称这个函数是双递归函数。

【例2-3】 Ackerman 函数

Ackerman 函数 $A(n,m)$ 有两个独立的整变量 $m \geqslant 0$ 和 $n \geqslant 0$，其定义如下：

$$\begin{cases} A(1,0) = 2 & \\ A(0,m) = 1 & m \geqslant 0 \\ A(n,0) = n + 2 & n \geqslant 2 \\ A(n,m) = A(A(n-1,m), m-1) & n,m \geqslant 1 \end{cases}$$

$A(n,m)$ 的自变量 m 的每一个值都定义了一个单变量函数。

由递归式的第 3 式知 $m = 0$ 定义了函数"加 2"。

当 $m = 1$ 时，由于 $A(1,1) = A(A(0,1),0) = A(1,0) = 2$ 以及 $A(n,1) = A(A(n-1,1),0) = A(n-1,1) + 2\ (n > 1)$，我们有 $A(n,1) = 2n(n \geqslant 1)$，即 $A(n,1)$ 是函数"乘 2"。

当 $m = 2$ 时，$A(n,2) = A(A(n-1,2),1) = 2A(n-1,2)$，和 $A(1,2) = A(A(0,2),1) = A(1,1) = 2$，故 $A(n,2) = 2^n$。

类似地可以推出，$A(n,3) = 2^{2^{\cdot^{\cdot^{2}}}}$，其中 2 的层数为 n。

$A(n,4)$ 的增长速度非常快，以至于没有适当的数学式子来表示这一函数。

单变量的 Ackerman 函数 $A(n)$ 定义为：$A(n) = A(n,n)$。其拟逆函数 $\alpha(n)$ 在算法复杂性分析中会遇到。它定义为：$\alpha(n) = \min\{k \mid A(k) \geqslant n\}$。即 $\alpha(n)$ 是使 $n \leqslant A(k)$ 成立的最小的 k 值。

例如，由 $A(0) = 1, A(1) = 2, A(2) = 4$ 和 $A(3) = 16$ 推知，$\alpha(1) = 0, \alpha(2) = 1, \alpha(3) = \alpha(4) = 2$ 和 $\alpha(5) = \cdots = \alpha(16) = 3$。可以看出 $\alpha(n)$ 的增长速度非常慢。

$A(4) = 2^{2^{\cdot^{\cdot^{2}}}}$（其中 2 的层数为 65 536）。这个数非常大，我们无法用通常的方式来表达它。如果要写出这个数将需要 $\log(A(4))$ 位，即 $2^{2^{\cdot^{\cdot^{2}}}}$（65 535 层 2 的方幂）那么多位。所以，对于通常所见到的正整数 n，我们有 $\alpha(n) \leqslant 4$。但在理论上 $\alpha(n)$ 没有上界，随着 n 的增加，它以难以想像的慢速度趋向正无穷大。

【例2-4】 排列问题

设 $R = \{r_1, r_2, \cdots, r_n\}$ 是要进行排列的 n 个元素，$R_i = R - \{r_i\}$。集合 X 中元素的全排列记为 $\mathrm{Perm}(X)$。$(r_i)\mathrm{Perm}(X)$ 表示在全排列 $\mathrm{Perm}(X)$ 的每一个排列前加上前缀 r_i 得到的排列。R 的全排列可归纳定义如下：

当 $n = 1$ 时，$\mathrm{Perm}(R) = (r)$，其中 r 是集合 R 中惟一的元素；

当 $n > 1$ 时，$\mathrm{Perm}(R)$ 由 $(r_1)\mathrm{Perm}(R_1), (r_2)\mathrm{Perm}(R_2), \cdots, (r_n)\mathrm{Perm}(R_n)$ 构成。

依此递归定义，可设计产生 $\mathrm{Perm}(R)$ 的递归算法如下：

```
template < class Type >
void Perm(Type list[ ], int k, int m)
{ // 产生 list[k:m] 的所有排列
      if (k = = m)
      { // 单元素序列
         for (int i = 0; i < = m; i + +)
            cout < < list[i];
         cout < < endl;
      }
      else // 多元素序列，递归产生排列
        for (int i = k; i < = m; i + +)
        {
           Swap(list[k], list[i]);
           Perm(list, k + 1, m);
```

```
            Swap(list[k], list[i]);
        }
}
template < class Type >
inline void Swap(Type & a, Type & b)
{
        Type temp = a; a = b; b = temp;
}
```

算法 Perm(list, k, m) 递归地产生所有前缀是 list[0: $k-1$],且后缀是 list[k: m] 的全排列的所有排列。函数调用 Perm(list, 0, $n-1$) 则产生 list[0: $n-1$] 的全排列。

在一般情况下,$k < m$。算法将 list[k: m] 中每一个元素分别与 list[k] 中元素交换。然后递归地计算 list[$k+1$: m] 的全排列,并将计算结果作为 list[0: k] 的后缀。算法中 Swap 是用于交换两个变量值的内联函数。

【例2-5】 整数划分问题

将一个正整数 n 表示成一系列正整数之和,

$$n = n_1 + n_2 + \cdots + n_k (其中, n_1 \geqslant n_2 \geqslant \cdots \geqslant n_k \geqslant 1, k \geqslant 1)$$

正整数 n 的一个这种表示称为正整数 n 的一个划分。正整数 n 的不同的划分个数称为正整数 n 的划分数,记作 $p(n)$。

例如,正整数 6 有如下 11 种不同的划分,所以 $p(6) = 11$。

6;

5 + 1;

4 + 2, 4 + 1 + 1;

3 + 3, 3 + 2 + 1, 3 + 1 + 1 + 1;

2 + 2 + 2, 2 + 2 + 1 + 1, 2 + 1 + 1 + 1 + 1;

1 + 1 + 1 + 1 + 1 + 1。

在正整数 n 的所有不同的划分中,将最大加数 n_1 不大于 m 的划分个数记作 $q(n, m)$。我们可以建立如下递归关系。

(1) $q(n, 1) = 1, n \geqslant 1$;

当最大加数 n_1 不大于 1 时,任何正整数 n 只有一种划分形式,即 $n = \overbrace{1 + 1 + \cdots + 1}^{n}$。

(2) $q(n, m) = q(n, n), m \geqslant n$;

最大加数 n_1 实际上不能大于 n。因此,$q(1, m) = 1$。

(3) $q(n, n) = 1 + q(n, n-1)$;

正整数 n 的划分由 $n_1 = n$ 的划分和 $n_1 \leqslant n-1$ 的划分组成。

(4) $q(n, m) = q(n, m-1) + q(n-m, m), n > m > 1$;

正整数 n 的最大加数 n_1 不大于 m 的划分由 $n_1 = m$ 的划分和 $n_1 \leqslant m-1$ 的划分组成。

以上的关系实际上给出了计算 $q(n, m)$ 的递归式如下:

$$q(n,m)=\begin{cases}1 & n=1,m=1\\ q(n,n) & n<m\\ 1+q(n,n-1) & n=m\\ q(n,m-1)+q(n-m,m) & n>m>1\end{cases}$$

据此，可设计计算 $q(n,m)$ 的递归函数如下。正整数 n 的划分数 $p(n)=q(n,n)$。

```
int q(int n, int m)
{
    if ((n < 1) || (m < 1)) return 0;
    if ((n == 1) || (m == 1)) return 1;
    if (n < m) return q(n,n);
    if (n == m) return q(n,m - 1) + 1;
    return q(n,m - 1) + q(n - m,m);
}
```

【例2-6】 Hanoi塔问题

设 A,B,C 是三个塔座。开始时，在塔座 A 上有一叠共 n 个圆盘，这些圆盘自下而上，由大到小地叠在一起，各圆盘从小到大编号为 $1,2,\cdots,n$，如图2-1所示。现要求将塔座 A 上的这一叠圆盘移到塔座 B 上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

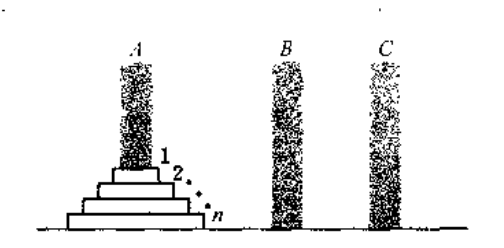


图 2-1 Hanoi塔问题的初始状态

规则(1) 每次只能移动一个圆盘；

规则(2) 任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

规则(3) 在满足移动规则(1)和(2)的前提下，可将圆盘移至 A,B,C 中任一塔座上。

这个问题有一个简单的解法。假设塔座 A、B、C 排成一个三角形，$A\to B\to C\to A$ 构成一顺时针循环。在移动圆盘的过程中，若是奇数次移动，则将最小的圆盘移到顺时针方向的下一塔座上；若是偶数次移动，则保持最小的圆盘不动。而在其他两个塔座之间，将较小的圆盘移到另一塔座上去。

上述算法简洁明确，可以证明它是正确的。下面我们用递归技术来解决同一问题。当 $n=1$ 时，问题比较简单。此时，只要将编号为1的圆盘从塔座 A 直接移至塔座 B 上即可。当 $n>1$ 时，需要利用塔座 C 作为辅助塔座。此时设法将 $n-1$ 个较小的圆盘依照移动规则从塔座 A 移至塔座 C，然后，将剩下的最大圆盘从塔座 A 移至塔座 B，最后，再设法将 $n-1$ 个较小的圆盘依照移动规则从塔座 C 移至塔座 B。这样一来，n 个圆盘的移动问题就可分解为两次 $n-1$ 个圆盘的移动问题，我们又可以递归地用上述方法来做。由此设计出解 Hanoi 塔问题的递归算法如下：

```
void Hanoi(int n, int A, int B, int C)
{
    if (n > 0) {
        Hanoi(n - 1,A,C,B);
        Move(n,A,B);
        Hanoi(n - 1,C,B,A);}
}
```

其中，Hanoi(n，A，B，C)表示将塔座 A 上自下而上，由大到小叠在一起的 n 个圆盘依移动规则移至塔座 B 上并仍按同样顺序叠排。在移动过程中，以塔座 C 作为辅助塔座。Move(n，A，B)表示将塔座 A 上编号为 n 的圆盘移至塔座 B 上。

算法 Hanoi 以递归形式给出，每个圆盘的具体移动方式并不清楚，因此很难用手工移动来模拟这个算法。然而，这个算法易于理解，也容易证明其正确性，且比通常的算法有效。

像 Hanoi 这样一个递归算法，在执行时需要多次调用自身。实现这种递归调用的关键是为算法建立递归调用工作栈。通常，在一个算法中调用另一算法时，系统需在运行被调用算法之前先完成三件事：

(1) 将所有实参指针，返回地址等信息传递给被调用算法；

(2) 为被调用算法的局部变量分配存储区；

(3) 将控制转移到被调用算法的入口。

在从被调用算法返回调用算法时，系统也相应地要完成三件事：

(1) 保存被调用算法的计算结果；

(2) 释放分配给被调用算法的数据区；

(3) 依照被调用算法保存的返回地址将控制转移到调用算法。

当有多个算法构成嵌套调用时，按照后调用先返回的原则进行。上述算法之间的信息传递和控制转移必须通过栈来实现，即系统将整个程序运行时所需的数据空间安排在一个栈中，每调用一个算法，就为它在栈顶分配一个存储区，每退出一个算法，就释放它在栈顶的存储区。当前正在运行的算法的数据一定在栈顶。

递归算法的实现类似于多个算法的嵌套调用，只是调用算法和被调用算法是同一个算法。和每次调用相关的一个重要概念是递归算法的调用层次。若调用一个递归算法的主算法为第0层算法，则从主算法调用递归算法为进入第1层调用；从第 i 层递归调用本算法为进入第 $i+1$ 层调用。反之，退出第 i 层递归调用，则返回至第 $i-1$ 层调用。为了保证递归调用正确执行，系统要建立一个递归调用工作栈，为各层次的调用分配数据存储区。每一层递归调用所需的信息构成一个工作记录，其中包括所有实参指针，所有局部变量以及返回上一层的地址。每进入一层递归调用，就产生一个新的工作记录压入栈顶。每退出一层递归调用，就从栈顶弹出一个工作记录。

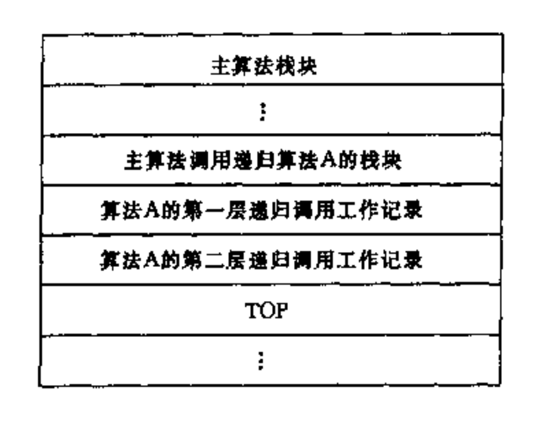


图 2-2　递归调用工作栈示意图

图 2-2 是实现算法递归调用的栈使用情况示意。其中 TOP 是指向栈顶的指针。

由于递归算法结构清晰，可读性强，且容易用数学归纳法证明算法的正确性，因此它为设计算法、调试程序带来很大方便。然而，递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。若在程序中消除算法的递归调用，则其运行时间可大为节省。因此，有时希望在一个递归算法中消除递归调用，使其转化为一个非递归算法。通常，消除递归是采用一个用户定义的栈来模拟系统的递归调用工作栈，从而达到将递归算法改为非递归算法的目的 仅仅是机械地模拟还不能达到减少计算时间和存储空间的目的，还需要根据具体程序的特点对递归调用工作栈进行简化，尽量减少栈操

作，压缩栈存储空间以达到节省计算时间和存储空间的目的。

2.2 分治法的基本思想

分治法的基本思想是将一个规模为 n 的问题分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题相同。递归地解这些子问题，然后将各子问题的解合并得到原问题的解。它的一般的算法设计模式如下：

```
Divide-and-Conquer(P)
  {
    if (|P|<= n0) Adhoc(P);
    divide P into smaller subinstances
    P1,P2,…,Pk;
    for (i = 1;i <= k;i++)
      yi = Divide-and-Conquer(Pi);
    return Merge(y1,y2,…,yk);
  }
```

其中，|P|表示问题P的规模，$n0$为一阈值，表示当问题P的规模不超过 $n0$时，问题已容易解出，不必再继续分解。Adhoc(P)是该分治法中的基本子算法，用于直接解小规模的问题 P。因此，当P的规模不超过 $n0$时，直接用算法 Adhoc(P)求解。算法 Merge($y1,y2,\cdots,yk$)是该分治法中的合并子算法，用于将 P 的子问题 P1，P2，…，Pk 的解 $y1,y2,\cdots,yk$ 合并为 P 的解。

根据分治法的分割原则，应把原问题分为多少个子问题才较适宜?每个子问题是否规模相同或怎样才为适当?这些问题很难予以肯定的回答。但人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。即将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。许多问题可以取 $k=2$。这种使子问题规模大致相等的做法是出自一种平衡(balancing)子问题的思想，它几乎总是比子问题规模不等的做法要好。

从分治法的一般设计模式可以看出，用它设计出的程序一般是一个递归算法。因此，分治法的计算效率通常可以用递归方程来进行分析。若一个分治法将规模为 n 的问题分成 k 个规模为n/m 的子问题。为方便起见，设分解阈值 $n0=1$，且 Adhoc 解规模为 1 的问题耗费 1 个单位时间，再设将原问题分解为 k 个子问题以及用 Merge 将 k 个子问题的解合并为原问题的解需用$f(n)$个单位时间。用 $T(n)$表示该分治法 Divide-and-Conquer(P)解规模为 $|P|=n$ 的问题所需的计算时间，则有：

$$T(n)=\begin{cases}O(1) & n=1\\ kT(n/m)+f(n) & n>1\end{cases}$$

下面讨论如何解这个与分治法有密切关系的递归方程。通常可以用展开递归式的方法来解这类递归方程，反复代入求解得

$$T(n)=n^{\log_m k}+\sum_{j=0}^{\log_m n-1}k^j f(n/m^j)$$

注意，递归方程及其解只给出 n 等于m 的方幂时 $T(n)$的值，但是如果认为 $T(n)$足够平滑，那么由 n 等于 m 的方幂时 $T(n)$ 的值可以估计 $T(n)$ 的增长速度。通常，我们可以假定 $T(n)$是单调上升的。

另一个需要注意的问题是:在分析分治法的计算效率时,通常得到的是递归不等式:

$$T(n) \leqslant \begin{cases} O(1) & n = n_0 \\ kT(n/m) + f(n) & n > n_0 \end{cases}$$

而我们关心的一般是最坏情况下的计算时间复杂度的上界,所以用等号(=)还是用小于或等于号(≤)是没有本质区别的。

以上讨论的是分治法的基本思想和一般原则。下面我们用一些具体例子来说明如何针对具体问题用分治思想来设计有效算法。

2.3 二分搜索技术

二分搜索算法是运用分治策略的典型例子。

给定已排好序的 n 个元素 a[0:$n-1$],现要在这 n 个元素中找出一特定元素 x。

首先较容易想到的是用顺序搜索方法,逐个比较a[0:$n-1$]中元素,直至找出元素 x 或搜索遍整个数组后确定 x 不在其中。这个方法没有很好地利用 n 个元素已排好序这个条件,因此在最坏情况下,顺序搜索方法需要 $O(n)$ 次比较。

二分搜索方法充分利用了元素间的次序关系,采用分治策略,可在最坏情况下用 $O(\log n)$ 时间完成搜索任务。

二分搜索算法的基本思想是将 n 个元素分成个数大致相同的两半,取 a[$n/2$] 与 x 作比较。如果 x = a[$n/2$],则找到 x,算法终止。如果 x < a[$n/2$],则我们只要在数组a的左半部继续搜索 x。如果 x > a[$n/2$],则我们只要在数组a的右半部继续搜索 x。具体算法可描述如下:

```
template < class Type >
int BinarySearch(Type a[], const Type& x, int n)
{// 在 a[0] < = a[1] < = ... < = a[n - 1] 中搜索 x
 // 找到 x 时返回其在数组中的位置,否则返回 - 1
    int left = 0; int right = n - 1;
    while (left < = right) {
      int middle = (left + right)/2;
      if (x = = a[middle]) return middle;
      if (x > a[middle]) left = middle + 1;
      else right = middle - 1;
      }
    return - 1; // 未找到 x
}
```

容易看出,每执行一次算法的 while 循环,待搜索数组的大小减少一半。因此,在最坏情况下,while 循环被执行了 $O(\log n)$ 次。循环体内运算需要 $O(1)$ 时间,因此整个算法在最坏情况下的计算时间复杂性为 $O(\log n)$。

二分搜索算法的思想易于理解,但是要写一个正确的二分搜索算法也不是一件简单的事。Knuth 在他的著作"The Art of Computer Programming: Sorting and Searching" 中提到,第一个二分搜索算法早在 1946 年就出现了,但是第一个完全正确的二分搜索算法却直到 1962 年才出现。Bentley 也在他的著作"Writing Correct Programs" 中写道,90% 的计算机专家不能在 2 小

时内写出完全正确的二分搜索算法。

2.4 大整数的乘法

通常,在分析一个算法的计算复杂性时,都将加法和乘法运算当作是基本运算来处理,即将执行一次加法或乘法运算所需的计算时间当作一个仅取决于计算机硬件处理速度的常数。这个假定仅在参加运算的整数能在计算机硬件对整数的表示范围内直接处理时才是合理的。然而,在某些情况下,我们要处理很大的整数,它无法在计算机硬件能直接表示的整数范围内进行处理。若用浮点数来表示它,则只能近似地表示它的大小,计算结果中的有效数字也受到限制。若要精确地表示大整数并在计算结果中要求精确地得到所有位数上的数字,就必须用软件的方法来实现大整数的算术运算。

设 X 和 Y 都是 n 位的二进制整数,现在要计算它们的乘积 XY。我们可以用小学所学的方法来设计一个计算乘积 XY 的算法,但是这样做计算步骤太多,显得效率较低。如果将每两个一位数的乘法或加法看作一步运算,那么这种方法要进行 $O(n^2)$ 步运算才能求出乘积 XY。下面我们用分治法来设计一个更有效的大整数乘积算法。

我们将 n 位的二进制整数 X 和 Y 都分为 2 段,每段的长为 $n/2$ 位(为简单起见,假设 n 是 2 的幂),如图 2-3 所示。

	$n/2$位	$n/2$位			$n/2$位	$n/2$位
$X=$	A	B		$Y=$	C	D

图 2-3 大整数 X 和 Y 的分段

由此,$X = A2^{n/2} + B$,$Y = C2^{n/2} + D$。这样,X 和 Y 的乘积为:

$$XY = (A2^{n/2} + B)(C2^{n/2} + D) = AC2^n + (AD + BC)2^{n/2} + BD$$

如果按此式计算 XY,则我们必须进行 4 次 $n/2$ 位整数的乘法(AC,AD,BC 和 BD),以及 3 次不超过 $2n$ 位的整数加法(分别对应于式中的加号),此外还要做 2 次移位(分别对应于式中乘 2^n 和乘 $2^{n/2}$)。所有这些加法和移位共用 $O(n)$ 步运算。设 $T(n)$ 是 2 个 n 位整数相乘所需的运算总数,则我们有:

$$T(n) = \begin{cases} O(1) & n = 1 \\ 4T(n/2) + O(n) & n > 1 \end{cases}$$

由此可得 $T(n) = O(n^2)$。因此,直接用此式来计算 X 和 Y 的乘积并不比小学生的方法更有效。要想改进算法的计算复杂性,必须减少乘法次数。下面我们把 XY 写成另一种形式:

$$XY = AC2^n + ((A - B)(D - C) + AC + BD)2^{n/2} + BD$$

此式看起来似乎更复杂些,但它仅需做 3 次 $n/2$ 位整数的乘法(AC,BD 和 $(A - B)(D - C)$),6 次加、减法和 2 次移位。由此可得:

$$T(n) = \begin{cases} O(1) & n = 1 \\ 3T(n/2) + O(n) & n > 1 \end{cases}$$

容易求得其解为 $T(n) = O(n^{\log 3}) = O(n^{1.59})$。这是一个较大的改进。

上述二进制大整数乘法同样可应用于十进制大整数的乘法以减少乘法次数,提高算法效率。如果将一个大整数分成 3 段或 4 段做乘法,计算复杂性会发生什么变化呢?是否优于分成 2 段来做乘法?读者可以通过有关练习得到明确的结论。

2.5 Strassen 矩阵乘法

矩阵乘法是线性代数中最常见的问题之一,它在数值计算中有广泛的应用。设 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 是两个 $n \times n$ 矩阵,它们的乘积 $\boldsymbol{AB}$ 同样是一个 $n \times n$ 矩阵。$\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的乘积矩阵 $\boldsymbol{C}$ 中元素 c_{ij} 定义为

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

若依此定义来计算 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的乘积矩阵 $\boldsymbol{C}$,则每计算 $\boldsymbol{C}$ 的一个元素 c_{ij},需要做 n 次乘法和 $n-1$ 次加法。因此,求出矩阵 $\boldsymbol{C}$ 的 n^2 个元素所需的计算时间为 $O(n^3)$。

20 世纪 60 年代末期,Strassen 采用了类似于我们在大整数乘法中用过的分治技术,将计算 2 个 n 阶矩阵乘积所需的计算时间改进到 $O(n^{\log 7}) = O(n^{2.81})$。其基本思想还是使用分治法。

首先,我们仍假设 n 是 2 的幂。将矩阵 $\boldsymbol{A}$,$\boldsymbol{B}$ 和 $\boldsymbol{C}$ 中每一矩阵都分块成为 4 个大小相等的子矩阵,每个子矩阵都是 $n/2 \times n/2$ 的方阵。由此可将方程 $\boldsymbol{C} = \boldsymbol{AB}$ 重写为

$$\begin{bmatrix} \boldsymbol{C}_{11} & \boldsymbol{C}_{12} \\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} \\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22} \end{bmatrix}$$

由此可得

$$\begin{aligned} \boldsymbol{C}_{11} &= \boldsymbol{A}_{11}\boldsymbol{B}_{11} + \boldsymbol{A}_{12}\boldsymbol{B}_{21} \\ \boldsymbol{C}_{12} &= \boldsymbol{A}_{11}\boldsymbol{B}_{12} + \boldsymbol{A}_{12}\boldsymbol{B}_{22} \\ \boldsymbol{C}_{21} &= \boldsymbol{A}_{21}\boldsymbol{B}_{11} + \boldsymbol{A}_{22}\boldsymbol{B}_{21} \\ \boldsymbol{C}_{22} &= \boldsymbol{A}_{21}\boldsymbol{B}_{12} + \boldsymbol{A}_{22}\boldsymbol{B}_{22} \end{aligned}$$

如果 $n = 2$,则 2 个 2 阶方阵的乘积可以直接计算出来,共需 8 次乘法和 4 次加法。当子矩阵的阶大于 2 时,为求 2 个子矩阵的积,可以继续将子矩阵分块,直到子矩阵的阶降为 2。这样,就产生了一个分治降阶的递归算法。依此算法,计算 2 个 n 阶方阵的乘积转化为计算 8 个 $n/2$ 阶方阵的乘积和 4 个 $n/2$ 阶方阵的加法。2 个 $n/2 \times n/2$ 矩阵的加法显然可以在 $O(n^2)$ 时间内完成。因此,上述分治法的计算时间耗费 $T(n)$ 应满足:

$$T(n) = \begin{cases} O(1) & n = 2 \\ 8T(n/2) + O(n^2) & n > 2 \end{cases}$$

这个递归方程的解仍然是 $T(n) = O(n^3)$。因此,该方法并不比用原始定义直接计算更有效。究其原因,是由于该方法并没有减少矩阵的乘法次数。而矩阵乘法耗费的时间要比矩阵加(减)法耗费的时间多得多。要想改进矩阵乘法的计算时间复杂性,必须减少乘法运算。

按照上述分治法的思想可以看出,要想减少乘法运算次数,关键在于计算 2 个 2 阶方阵的乘积时,所用乘法次数能否少于 8 次。Strassen 提出了一种新的算法来计算 2 个 2 阶方阵的乘积。他的算法只用了 7 次乘法运算,但增加了加、减法的运算次数。这 7 次乘法是:

$$\begin{aligned} \boldsymbol{M}_1 &= \boldsymbol{A}_{11}(\boldsymbol{B}_{12} - \boldsymbol{B}_{12}) \\ \boldsymbol{M}_2 &= (\boldsymbol{A}_{11} + \boldsymbol{A}_{12})\boldsymbol{B}_{22} \\ \boldsymbol{M}_3 &= (\boldsymbol{A}_{21} + \boldsymbol{A}_{22})\boldsymbol{B}_{11} \\ \boldsymbol{M}_4 &= \boldsymbol{A}_{22}(\boldsymbol{B}_{21} - \boldsymbol{B}_{11}) \\ \boldsymbol{M}_5 &= (\boldsymbol{A}_{11} + \boldsymbol{A}_{22})(\boldsymbol{B}_{11} + \boldsymbol{B}_{22}) \end{aligned}$$

$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

做了这7次乘法后，再做若干次加、减法就可以得到：

$$C_{11} = M_5 + M_4 - M_2 + M_6$$
$$C_{12} = M_1 + M_2$$
$$C_{21} = M_3 + M_4$$
$$C_{22} = M_5 + M_1 - M_3 - M_7$$

以上计算的正确性很容易验证。

Strassen矩阵乘积分治算法中，用了7次对于$n/2$阶矩阵乘积的递归调用和18次$n/2$阶矩阵的加减运算。由此可知，该算法的所需的计算时间$T(n)$满足如下的递归方程：

$$T(n) = \begin{cases} O(1) & n = 2 \\ 7T(n/2) + O(n^2) & n > 2 \end{cases}$$

解此递归方程得$T(n) = O(n^{\log 7}) \approx O(n^{2.81})$。由此可见，Strassen矩阵乘法的计算时间复杂性比普通矩阵乘法有较大改进。

有人曾列举了计算2个2×2阶矩阵乘法的36种不同方法。但所有的方法都至少做7次乘法。除非能找到一种计算2阶方阵乘积的算法，使乘法的计算次数少于7次，计算矩阵乘积的计算时间下界才有可能低于$O(n^{2.81})$。但是Hopcroft和Kerr已经证明(1971)，计算2个2×2矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究3×3或5×5矩阵的更好算法。在Strassen之后又有许多算法改进了矩阵乘法的计算时间复杂性，目前最好的计算时间上界是$O(n^{2.376})$。而目前所知道的矩阵乘法的最好下界仍是它的平凡下界$\Omega(n^2)$。因此到目前为止还无法确切知道矩阵乘法的时间复杂性。关于这一研究课题还有许多工作可做。

2.6 棋盘覆盖

在一个$2^k \times 2^k$个方格组成的棋盘中，若恰有一个方格与其他方格不同，则称该方格为一特殊方格，且称该棋盘为一特殊的棋盘。显然特殊方格在棋盘上出现的位置有4^k种情形。因而对任何$k \geq 0$，有4^k种不同的特殊棋盘。图2-4中的特殊棋盘是当$k = 2$时16个特殊棋盘中的一个。

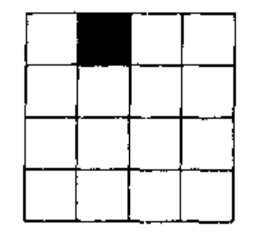


图2-4　$k = 2$时的一个特殊棋盘

在棋盘覆盖问题中，我们要用图2-5所示的4种不同形态的L型骨牌覆盖一个给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。易知，在任何一个$2^k \times 2^k$的棋盘覆盖中，用到的L型骨牌个数恰为$(4^k - 1)/3$。

用分治策略，我们可以设计出解棋盘覆盖问题的一个简捷的算法

当$k > 0$时，我们将$2^k \times 2^k$棋盘分割为4个$2^{k-1} \times 2^{k-1}$子棋盘如图2-6(a)所示。

特殊方格必位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。为了将这3个无特殊方格的子棋盘转化为特殊棋盘，我们可以用一个L型骨牌覆盖这3个较小棋盘的会合处，如图2-6(b)所示，这3个子棋盘上被L型骨牌覆盖的方格就成为该棋盘上的特殊方格，从而将

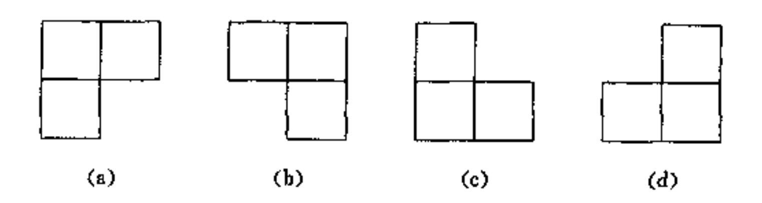
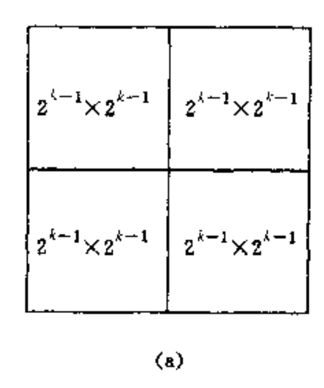


图 2-5　4 种不同形态的 L 型骨牌



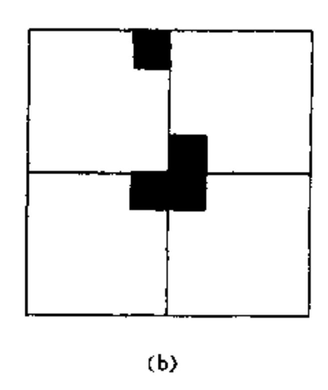


图 2-6　棋盘分割

原问题转化为 4 个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为 1 × 1 棋盘。

实现这种分治策略的算法 ChessBoard 可实现如下：

```
void ChessBoard(int tr, int tc, int dr, int dc, int size)
{
    if (size == 1) return;
    int t = tile++, // L 型骨牌号
        s = size/2; // 分割棋盘

    // 覆盖左上角子棋盘
    if (dr < tr + s && dc < tc + s)
        // 特殊方格在此棋盘中
        ChessBoard(tr, tc, dr, dc, s);
    else {// 此棋盘中无特殊方格
          // 用 t 号 L 型骨牌覆盖右下角
          Board[tr + s - 1][tc + s - 1] = t;
          // 覆盖其余方格
          ChessBoard(tr, tc, tr + s - 1, tc + s - 1, s);}

    // 覆盖右上角子棋盘
    if (dr < tr + s && dc >= tc + s)
        // 特殊方格在此棋盘中
        ChessBoard(tr, tc + s, dr, dc, s);
    else {// 此棋盘中无特殊方格
          // 用 t 号 L 型骨牌覆盖左下角
          Board[tr + s - 1][tc + s] = t;
```

```
            // 覆盖其余方格
            ChessBoard(tr, tc + s, tr + s - 1, tc + s, s);}

    // 覆盖左下角子棋盘
    if (dr >= tr + s && dc < tc + s)
        // 特殊方格在此棋盘中
        ChessBoard(tr + s, tc, dr, dc, s);
    else {// 用 t 号 L 型骨牌覆盖右上角
        Board[tr + s][tc + s - 1] = t;
        // 覆盖其余方格
        ChessBoard(tr + s, tc, tr + s, tc + s - 1, s);}

    // 覆盖右下角子棋盘
    if (dr >= tr + s && dc >= tc + s)
        // 特殊方格在此棋盘中
        ChessBoard(tr + s, tc + s, dr, dc, s);
    else {// 用 t 号 L 型骨牌覆盖左上角
            Board[tr + s][tc + s] = t;
            // 覆盖其余方格
            ChessBoard(tr + s, tc + s, tr + s, tc + s, s);}
}
```

上述算法中,用一个二维整型数组 Board 表示棋盘。Board[0][0] 是棋盘的左上角方格。tile 是算法中的一个全局整型变量,用来表示 L 型骨牌的编号,其初始值为 0。算法的输入参数是:

tr: 棋盘左上角方格的行号;

tc: 棋盘左上角方格的列号;

dr: 特殊方格所在的行号;

dc: 特殊方格所在的列号;

size: size $= 2^k$,棋盘规格为 $2^k \times 2^k$。

设 $T(k)$ 是算法 ChessBoard 覆盖一个 $2^k \times 2^k$ 棋盘所需的时间,则从算法的分治策略可知,$T(k)$ 满足如下递归方程:

$$T(k) = \begin{cases} O(1) & k = 0 \\ 4T(k-1) + O(1) & k > 0 \end{cases}$$

解此递归方程可得 $T(k) = O(4^k)$。由于覆盖一个 $2^k \times 2^k$ 棋盘所需的 L 型骨牌个数为 $(4^k - 1)/3$,故算法 ChessBoard 是一个在渐近意义下最优的算法。

2.7 合并排序

合并排序算法是用分治策略实现对 n 个元素进行排序的算法。其基本思想是:当 $n = 1$ 时终止排序,否则将待排序元素分成大小大致相同的两个子集合,分别对两个子集合进行排序,最终将排好序的子集合合并成为所要求的排好序的集合。合并排序算法可递归地描述如下:

```
template <class Type>
void MergeSort(Type a[], int left, int right)
```

```
{
    if (left < right) {// 至少有 2 个元素
    int i = (left + right)/2; // 取中点
    MergeSort(a, left, i);
    MergeSort(a, i + 1, right);
    Merge(a, b, left, i, right); // 合并到数组 b
    Copy(a, b, left, right); // 复制回数组 a
    }
}
```

其中,算法 Merge合并两个排好序的数组段到一个新的数组b中,然后由Copy将合并后的数组段再复制回数组a中。Merge和Copy显然可在 $O(n)$ 时间内完成,因此合并排序算法对 n 个元素进行排序,在最坏情况下所需的计算时间 $T(n)$ 满足

$$T(n) = \begin{cases} O(1) & n \leqslant 1 \\ 2T(n/2) + O(n) & n > 1 \end{cases}$$

解此递归方程可知 $T(n) = O(n\log n)$。由于排序问题的计算时间下界为 $\Omega(n\log n)$,故合并排序算法是一个渐近最优算法

对于算法 MergeSort,还可以从多方面对它进行改进。例如,从分治策略的机制入手,容易消除算法中的递归。事实上,算法 MergeSort 的递归过程只是将待排序集合一分为二,直至待排序集合只剩下一个元素为止,然后不断合并两个排好序的数组段。按此机制,我们可以首先将数组a中相邻元素两两配对。用合并算法将它们排序,构成 $n/2$ 组长度为2的排好序的子数组段,然后再将它们排序成长度为4的排好序的子数组段,如此继续下去,直至整个数组排好序。

按此思想,消去递归后的合并排序算法可描述如下:

```
template < class Type >
void MergeSort(Type a[], int n)
{
    Type *b = new Type [n];
    int s = 1;
    while (s < n) {
      MergePass(a, b, s, n); // 合并到数组 b
      s += s;
      MergePass(b, a, s, n); // 合并到数组 a
      s += s;
      }
}
```

其中,函数 MergePass用于合并排好序的相邻数组段。具体的合并算法由 Merge来实现。其中要注意定义关于类型为 Type 的元素的比较运算"<="。特别地,如果 Type 是自定义的,则必须重载运算"<="。

```
template < class Type >
void MergePass(Type x[], Type y[], int s, int n)
{// 合并大小为 s 的相邻子数组
    int i = 0;
```

```
    while (i <= n - 2 * s) {
        // 合并大小为s的相邻2段子数组
        Merge(x, y, i, i + s - 1, i + 2 * s - 1);
        i = i + 2 * s;
    }

    // 剩下的元素个数少于2s
    if (i + s < n) Merge(x, y, i, i + s - 1, n - 1);
    else for (int j = i; j <= n - 1; j++)
             y[j] = x[j];
}

template <class Type>
void Merge(Type c[], Type d[], int l, int m, int r)
{// 合并c[l:m]和c[m+1:r]到d[l:r]
    int i = l,
        j = m + 1,
        k = l;
    while ((i <= m) && (j <= r))
        if (c[i] <= c[j]) d[k++] = c[i++];
        else d[k++] = c[j++];
    if (i > m) for (int q = j; q <= r; q++)
                   d[k++] = c[q];
    else for (int q = i; q <= m; q++)
             d[k++] = c[q];
}
```

自然合并排序是上述合并排序算法MergeSort的一个变形。在上述合并排序算法中，我们在第一步合并相邻长度为1的子数组段，这是因为长度为1的子数组段是已排好序的。事实上，对于初始给定的数组a，通常存在多个长度大于1的已自然排好序的子数组段。例如，若数组a中元素为{4,8,3,7,1,5,6,2}，则自然排好序的子数组段有{4,8}，{3,7}，{1,5,6}和{2}。用1次对数组a的线性扫描就足以找出所有这些排好序的子数组段。然后将相邻的排好序的子数组段两两合并，构成更大的排好序的子数组段。对上面的例子，经一次合并后我们得到2个合并后的子数组段{3,4,7,8}和{1,2,5,6}。继续合并相邻排好序的子数组段，直至整个数组已排好序。上面这2个数组段再合并后就得到{1,2,3,4,5,6,7,8}。

上述思想就是自然合并排序算法的基本思想。在通常情况下，按此方式进行合并排序所需的合并次数较少。例如，对于所给的n元素数组已排好序的极端情况，自然合并排序算法不需要执行合并步，而算法MergeSort需要执行$\lceil \log n \rceil$次合并。因此，在这种情况下，自然合并排序算法需要$O(n)$时间，而算法MergeSort需要$O(n\log n)$时间。

2.8 快速排序

快速排序算法是基于分治策略的另一个排序算法。其基本思想是，对于输入的子数组

a[$p:r$],按以下三个步骤进行排序:

(1) 分解(Divide):以 a[p] 为基准元素将 a[$p:r$] 划分成 3 段 a[$p:q-1$],a[q] 和 a[$q+1:r$],使得 a[$p:q-1$] 中任何一个元素小于等于 a[q],a[$q+1:r$] 中任何一个元素大于等于 a[q]。下标 q 在划分过程中确定。

(2) 递归求解(Conquer):通过递归调用快速排序算法分别对 a[$p:q-1$] 和 a[$q+1:r$] 进行排序。

(3) 合并(Merge):由于对 a[$p:q-1$] 和 a[$q+1:r$] 的排序是就地进行的,所以在 a[$p:q-1$] 和 a[$q+1:r$] 都已排好的序后不需要执行任何计算 a[$p:r$] 就已排好序。

基于这个思想,可实现快速排序算法如下:

```
template < class Type >
void QuickSort (Type a[ ], int p, int r)
{
    if (p < r) {
    int q = Partition(a,p,r);
    QuickSort (a,p,q - 1); //对左半段排序
    QuickSort (a,q + 1,r); //对右半段排序
    }
}
```

对含有 n 个元素的数组 a[$0:n-1$] 进行快速排序只要调用 QuickSort(a,0,$n-1$) 即可。

上述算法中的函数 Partition,以一个确定的基准元素 a[p] 对子数组 a[$p:r$] 进行划分,它是快速排序算法的关键。

```
template < class Type >
int Partition (Type a[ ], int p, int r)
{
    int i = p,
       j = r + 1;
    Type x = a[p];
    // 将 < x 的元素交换到左边区域
    // 将 > x 的元素交换到右边区域
    while (true) {
       while (a[++i] < x);
       while (a[--j] > x);
       if (i >= j) break;
       Swap(a[i], a[j]);
       }
    a[p] = a[j];
    a[j] = x;
    return j;
}
```

Partition 对 a[$p:r$] 进行划分时,以元素 x = a[p] 作为划分的基准,分别从左、右两端开始,扩展两个区域 a[$p:i$] 和 a[$j:r$],使得 a[$p:i$] 中元素小于或等于 x,而 a[$j:r$] 中元素大于或等于 x。初始时,$i=p$,且 $j=r+1$。

在 while 循环体中,下标 j 逐渐减小,i 逐渐增大,直到 a[i] $\geqslant x \geqslant$ a[j]。如果这两个不等

式是严格的，则 a[i] 不会是左边区域的元素，而 a[j] 不会是右边区域的元素。此时若 $i < j$，就应该交换 a[i] 与 a[j] 的位置，扩展左右两个区域。

while循环重复至 $i \geqslant j$ 时结束。这时 a[$p:r$] 已被划分成 a[$p:q-1$]，a[q] 和 a[$q+1:r$]，且满足 a[$p:q-1$] 中元素不大于 a[$q+1:r$] 中元素。在 Partition 结束时返回划分点 $q = j$。

事实上，函数 Partition 的主要功能就是将小于 x 的元素放在原数组的左半部分。而将大于 x 的元素放在原数组的右半部分。其中，有一些细节需要注意。例如，算法中的下标 i 和 j 不会超出 a[$p:r$] 的下标界。另外，在快速排序算法中选取 a[p] 作为基准可以保证算法正常结束。如果选择 a[r] 作为划分的基准，且 a[r] 又是 a[$p:r$] 中的最大元素，则 Partition 返回的值为 $q = r$，这就会使 QuickSort 陷入死循环。

对于输入序列 a[$p:r$]，Partition 的计算时间显然为 $O(r-p-1)$。

快速排序的运行时间与划分是否对称有关，其最坏情况发生在划分过程产生的两个区域分别包含 $n-1$ 个元素和1个元素的时候。由于函数 Partition 的计算时间为 $O(n)$，所以如果算法 Partition 的每一步都出现这种不对称划分，则其计算时间复杂性 $T(n)$ 满足

$$T(n) = \begin{cases} O(1) & n \leqslant 1 \\ T(n-1) + O(n) & n > 1 \end{cases}$$

解此递归方程可得 $T(n) = O(n^2)$。

在最好情况下，每次划分所取的基准都恰好为中值，即每次划分都产生两个大小为 $n/2$ 的区域，此时，Partition 的计算时间 $T(n)$ 满足

$$T(n) = \begin{cases} O(1) & n \leqslant 1 \\ 2T(n/2) + O(n) & n > 1 \end{cases}$$

其解为 $T(n) = O(n\log n)$。

可以证明，快速排序算法在平均情况下的时间复杂性也是 $O(n\log n)$，这在基于比较的排序算法类中算是快速的，快速排序也因此而得名。

我们已看到，快速排序算法的性能取决于划分的对称性。通过修改函数 Partition，可以设计出采用随机选择策略的快速排序算法。在快速排序算法的每一步中，当数组还没有被划分时，可以在 a[$p:r$] 中随机选出一个元素作为划分基准，这样可以使划分基准的选择是随机的，从而可以期望划分是较对称的。随机化的划分算法可实现如下：

```
template < class Type >
int RandomizedPartition (Type a[], int p, int r)
{
     int i = Random(p,r);
     Swap(a[i], a[p]);
     return Partition (a, p, r);
}
```

其中，函数 Random(p,r) 产生 p 和 r 之间的一个随机整数，且产生不同整数的概率相同。随机化的快速排序算法通过调用 RandomizedPartition 来产生随机的划分。

```
template < class Type >
void RandomizedQuickSort (Type a[], int p, int r)
{
     if (p < r) {
```

```
        int q = RandomizedPartition(a,p,r);
        RandomizedQuickSort (a,p,q - 1); // 对左半段排序
        RandomizedQuickSort (a,q + 1,r); // 对右半段排序
        }
}
```

2.9 线性时间选择

在这一节中,我们要讨论与排序问题类似的元素选择问题。元素选择问题的一般提法是:给定线性序集中 n 个元素和一个整数 k,$1 \leqslant k \leqslant n$ 要求找出这 n 个元素中第 k 小的元素,即如果将这 n 个元素依其线性序排列时,排在第 k 个位置的元素即为我们要找的元素。当 $k = 1$ 时,就是要找最小元素;当 $k = n$ 时,就是要找最大元素;当 $k = (n + 1)/2$ 时,称为找中位数。

在某些特殊情况下,很容易设计出解选择问题的线性时间算法。例如,找 n 个元素的最小元素和最大元素显然可以在 $O(n)$ 时间完成。如果 $k \leqslant n/\log n$,通过堆排序算法可以在 $O(n + k\log n) = O(n)$ 时间内找出第 k 小元素。当 $k \geqslant n - n/\log n$ 时也一样。

一般的选择问题,特别是中位数的选择问题似乎比找最小元素要难。但事实上,从渐近阶的意义上看,它们是一样的。一般的选择问题也可以在 $O(n)$ 时间内得到解决。下面我们讨论解一般选择问题的一个分治算法 RandomizedSelect,该算法实际上是模仿快速排序算法设计出来的。其基本思想也是对输入数组进行递归划分。与快速排序算法不同的是,它只对划分出的子数组之一进行递归处理。

算法 RandomizedSelect 用到我们在随机快速排序算法中讨论过的随机划分函数 RandomizedPartition。因此,划分是随机地产生的。由此导致算法 RandomizedSelect 也是一个随机化的算法。要找数组 $a[0:n-1]$ 中第 k 小元素只要调用 RandomizedSelect(a,0,$n-1$,k) 即可。具体算法可描述如下:

```
template < class Type >
Type RandomizedSelect(Type a[ ],int p,int r,int k)
{
    if (p = = r) return a[p];
    int i = RandomizedPartition(a,p,r),
      j = i - p + 1;
    if (k < = j) return RandomizedSelect(a,p,i,k);
    else return RandomizedSelect(a,i + 1,r,k - j);
}
```

在算法 RandomizedSelect 中执行 RandomizedPartition 后,数组 $a[p:r]$ 被划分成两个子数组 $a[p:i]$ 和 $a[i+1:r]$,使得 $a[p:i]$ 中每个元素都不大于 $a[i+1:r]$ 中每个元素。接着算法计算子数组 $a[p:i]$ 中元素个数 j。如果 $k \leqslant j$,则 $a[p:r]$ 中第 k 小元素落在子数组 $a[p:i]$ 中。如果 $k > j$,则要找的第 k 小元素落在子数组 $a[i+1:r]$ 中。由于此时已知道子数组 $a[p:i]$ 中元素均小于要找的第 k 小元素,因此,要找的 $a[p:r]$ 中第 k 小元素是 $a[i+1:r]$ 中的第 $k-j$ 小元素。

容易看出,在最坏情况下,算法 RandomizedSelect 需要 $\Omega(n^2)$ 计算时间。例如在找最小元素时,总是在最大元素处划分。尽管如此,该算法的平均性能很好。

由于随机划分函数 RandomizedPartition 使用了一个随机数产生器 Random，它能随机地产生 p 和 r 之间的一个随机整数，因此，RandomizedPartition 产生的划分基准是随机的。在这个条件下，可以证明，算法 RandomizedSelect 可以在 $O(n)$ 平均时间内找出 n 个输入元素中的第 k 小元素。

下面来讨论一个类似于 RandomizedSelect 但可以在最坏情况下用 $O(n)$ 时间就完成选择任务的算法 Select。如果我们能在线性时间内找到一个划分基准，使得按这个基准所划分出的两个子数组的长度都至少为原数组长度的 ε 倍（$0 < \varepsilon < 1$ 是某个正常数），那么在最坏情况下用 $O(n)$ 时间就可以完成选择任务。例如，若 $\varepsilon = 9/10$，算法递归调用所产生的子数组的长度至少缩短 1/10。所以，在最坏情况下，算法所需的计算时间 $T(n)$ 满足递归式 $T(n) \leqslant T(9n/10) + O(n)$。由此可得 $T(n) = O(n)$。

我们可以按以下步骤来寻找一个好的划分基准：

（1）将 n 个输入元素划分成 $\lceil n/5 \rceil$ 个组，每组 5 个元素，除可能有一个组不是 5 个元素外。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共 $\lceil n/5 \rceil$ 个。

（2）递归调用 Select 来找出这 $\lceil n/5 \rceil$ 个元素的中位数。如果 $\lceil n/5 \rceil$ 是偶数，就找它的两个中位数中较大的一个。然后以这个元素作为划分基准。

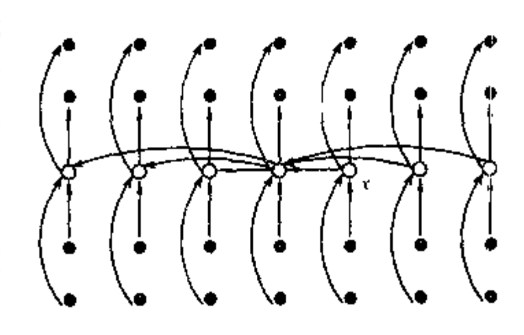


图 2-7　选择划分基准

图 2-7 是上述划分策略的示意图，其中 n 个元素用小圆点来表示，空心小圆点为每组元素的中位数。中位数的中位数 x 在图中标出。图中所画箭头是由较大元素指向较小元素。

只要等于基准的元素不太多，利用这个基准来划分的两个子数组的大小就不会相差太远。为了简化问题，我们先设所有元素互不相同。在这种情况下，找出的基准 x 至少比 $3\lfloor (n-5)/10 \rfloor$ 个元素大，因为在每一组中有两个元素小于本组的中位数，而 $\lfloor n/5 \rfloor$ 个中位数中又有 $\lfloor (n-5)/10 \rfloor$ 个小于基准 x。同理，基准 x 也至少比 $3\lfloor (n-5)/10 \rfloor$ 个元素小。而当 $n \geqslant 75$ 时，$3\lfloor (n-5)/10 \rfloor \geqslant n/4$。所以按此基准划分所得的两个子数组的长度都至少缩短 1/4。这一点是至关重要的。据此，我们给出算法 Select 如下：

```
template < class Type >
Type Select( Type a[ ], int p, int r, int k)
{
    if (r - p < 75) {
       用某个简单排序算法对数组 a[p:r] 排序;
       return a[p + k - 1];
     };
    for ( int i = 0; i < = (r - p - 4)/5; i + + )
       将 a[p + 5 * i] 至 a[p + 5 * i + 4] 的第 3 小元素
       与 a[p + i] 交换位置;
    // 找中位数的中位数，r - p - 4 即上面所说的 n - 5
    Type x = Select(a, p, p + (r - p - 4)/5, (r - p - 4)/10);
    int i = Partition(a,p,r, x),
        j = i - p + 1;
```

```
    if (k <= j) return Select(a,p,i,k);
    else return Select(a,i+1,r,k-j);
}
```

为了分析算法 Select 的计算时间复杂性,设 $n = r - p + 1$,即 n 为输入数组的长度。算法的递归调用只有在 $n \geq 75$ 时才执行。因此,当 $n < 75$ 时算法 Select 所用的计算时间不超过一个常数 C_1。找到中位数的中位数 x 后,算法 Select 以 x 为划分基准调用函数 Partition 对数组 a[$p:r$] 进行划分,这需要 $O(n)$ 时间。算法 Select 的 for 循环体行共执行 $n/5$ 次,每一次需要 $O(1)$ 时间。因此,执行 for 循环共需 $O(n)$ 时间。

设对 n 个元素的数组调用 Select 需要 $T(n)$ 时间,那么找中位数的中位数 x 至多用了 $T(n/5)$ 的时间。我们已经证明了,按照算法所选的基准 x 进行划分所得到的两个子数组分别至多有 $3n/4$ 个元素。所以无论对哪一个子数组调用 Select 都至多用了 $T(3n/4)$ 的时间。

由此,我们得到关于 $T(n)$ 的递归式

$$T(n) \leq \begin{cases} C_1 & n < 75 \\ C_2 n + T(n/5) + T(3n/4) & n \geq 75 \end{cases}$$

解此递归式可得 $T(n) = O(n)$。

由于我们将每一组的大小定为5,并选取75作为是否作递归调用的分界点。这两点保证了 $T(n)$ 的递归式中两个自变量之和 $n/5 + 3n/4 = 19n/20 = \alpha n, 0 < \alpha < 1$。这是使 $T(n) = O(n)$ 的关键之处。当然,除了 5 和 75 之外,我们还可以有其他选择。

在算法 Select 中,我们假设所有元素互不相等,这是为了保证在以 x 为划分基准调用函数 Partition 对数组 a[$p:r$] 进行划分之后,所得到的两个子数组的长度都不超过原数组长度的 3/4。当元素可能相等时,应在划分之后加一个语句,将所有与基准 x 相等的元素集中在一起,如果这样的元素的个数 $m \geq 1$,而且 $j \leq k \leq j + m - 1$ 时,就不必再递归调用,只要返回 a[i] 即可。否则最后一行改为调用 Select($i + m + 1, r, k - j - m$)。

2.10 最接近点对问题

在计算机应用中,常用诸如点、圆等简单的几何对象表达现实世界中的实体。在涉及这些几何对象的问题中,常需要了解其邻域中其他几何对象的信息。例如,在空中交通控制问题中,若将飞机作为空间中移动的一个点来处理,则具有最大碰撞危险的两架飞机,就是这个空间中最接近的一对点。这类问题是计算几何学中研究的基本问题之一。下面我们着重考虑平面上的最接近点对问题。

最接近点对问题的提法是:给定平面上 n 个点,找其中的一对点,使得在 n 个点组成的所有点对中,该点对间的距离最小。

严格地说,最接近点对可能多于一对,为简单起见,我们只找其中的一对作为问题的解。这个问题似乎很容易解决。我们只要将每一点与其他 $n-1$ 个点的距离算出,找出达到最小距离的两点即可。然而,这样做效率太低,需要 $O(n^2)$ 的计算时间。可以证明,该问题的计算时间下界为 $\Omega(n\log n)$。这个下界引导我们去找问题的一个 $\theta(n\log n)$ 时间算法。很自然地我们会想到用分治法来解这个问题。

将所给的平面上 n 个点的集合 S 分成两个子集 S_1 和 S_2,每个子集中约有 $n/2$ 个点。然后在每个子集中递归地求其最接近的点对。在这里,一个关键的问题是如何实现分治法中的合并

步骤，即由 S_1 和 S_2 的最接近点对，如何求得原集合 S 中的最接近点对。如果组成 S 的最接近点对的两个点都在 S_1 中或都在 S_2 中，则问题很容易解决。但是，如果这两个点分别在 S_1 和 S_2 中，则问题就复杂些。

为使问题易于理解和分析，我们先来考虑一维的情形。此时，S 中的 n 个点退化为 x 轴上的 n 个实数 $x_1, x_2, \cdots, x_n$。最接近点对即为这 n 个实数中相差最小的两个实数。显然可以先将 $x_1, x_2, \cdots, x_n$ 排好序，然后，用一次线性扫描就可以找出最接近点对。这种方法的主要计算时间花在排序上，因此耗时 $O(n\log n)$。然而这种方法无法直接推广到二维的情形。因此，对这种一维的简单情形，我们还是尝试用分治法来求解，并希望推广到二维的情形。

假设我们用 x 轴上某个点 m 将 S 划分为两个集合 S_1 和 S_2，使得 $S_1 = \{x \in S \mid x \leqslant m\}$；$S_2 = \{x \in S \mid x > m\}$。这样一来，对于所有 $p \in S_1$ 和 $q \in S_2$ 有 $p < q$。

递归地在 S_1 和 S_2 上找出其最接近点对 $\{p_1, p_2\}$ 和 $\{q_1, q_2\}$，并设

$$d = \min\{|p_1 - p_2|, |q_1 - q_2|\}$$

由此易知，S 中的最接近点对或者是 $\{p_1, p_2\}$，或者是 $\{q_1, q_2\}$，或者是某个 $\{p_3, q_3\}$，其中，$p_3 \in S_1$ 且 $q_3 \in S_2$。如图 2-8 所示。

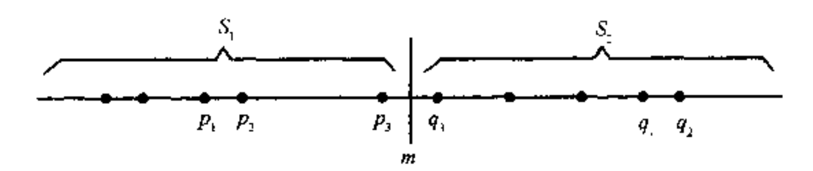


图 2-8　一维情形的分治法

我们注意到，如果 S 的最接近点对是 $\{p_3, q_3\}$，即 $|p_3 - q_3| < d$，则 p_3 和 q_3 两者与 m 的距离不超过 d，即 $|p_3 - m| < d$，$|q_3 - m| < d$。也就是说，$p_3 \in (m - d, m]$，$q_3 \in (m, m + d]$。由于每个长度为 d 的半闭区间至多包含 S_1 中的一个点，并且 m 是 S_1 和 S_2 的分割点，因此 $(m - d, m]$ 中至多包含一个 S 中的点。同理，$(m, m + d]$ 中也至多包含一个 S 中的点。由图 2-8 可以看出，如果 $(m - d, m]$ 中有 S 中点，则此点就是 S_1 中最大点。同理，如果 $(m, m + d]$ 中有 S 中的点，则此点就是 S_2 中最小点。因此，我们用线性时间就能找到区间 $(m - d, m]$ 和 $(m, m + d]$ 中所有点，即 p_3 和 q_3。从而我们用线性时间就可以将 S_1 的解和 S_2 的解合并成为 S 的解。也就是说，按这种分治策略，合并步可在 $O(n)$ 时间内完成。这样是否就可以得到一个有效的算法呢？还有一个问题需要认真考虑，即分割点 m 的选取，及 S_1 和 S_2 的划分。选取分割点 m 的一个基本要求是由此导出集合 S 的一个线性分割，即 $S = S_1 \cup S_2$，$S_1 \neq \varnothing$，$S_2 \neq \varnothing$，且 $S_1 \subset \{x \mid x \leqslant m\}$，$S_2 \subset \{x \mid x > m\}$。容易看出，如果选取 $m = \frac{\max(S) + \min(S)}{2}$，可以满足线性分割的要求。选取分割点后，再用 $O(n)$ 时间即可将 S 划分成 $S_1 = \{x \in S \mid x \leqslant m\}$ 和 $S_2 = \{x \in S \mid x > m\}$。然而，这样选取分割点 m，有可能造成划分出的子集 S_1 和 S_2 的不平衡。例如在最坏情况下，$|S_1| = 1$，$|S_2| = n - 1$，由此产生的分治法在最坏情况下所需的计算时间 $T(n)$ 应满足递归方程：

$$T(n) = T(n - 1) + O(n)$$

它的解是 $T(n) = O(n^2)$。这种效率降低的现象可以通过分治法中“平衡子问题”的方法加以解决。我们可以适当选择分割点 m，使 S_1 和 S_2 中有个数大致相等的点。自然地，我们会想到用 S 中各点坐标的中位数来作分割点。Blum 的选取中位数的线性时间算法使我们可以在 $O(n)$

时间内确定一个平衡的分割点 m

至此,我们设计出一个求一维点集 S 的最接近点对的算法 Cpair1 如下:

```
bool Cpair1(S,d)
{
    n = |S|;
    if (n < 2) {
        d = ∞;
        return false;
        }
    m = S中各点坐标的中位数;
    构造 S1 和 S2;
    //S1 = {x ∈ S | x <= m}, S2 = {x ∈ S | x > m}
    Cpair1(S1,d1);
    Cpair1(S2,d2);
    p = max(S1);
    q = min(S2);
    d = min(d1,d2,q - p);
    return true;
}
```

以上分析可知,该算法的分割步骤和合并步骤总共耗时 $O(n)$。因此,算法耗费的计算时间 $T(n)$ 满足递归方程

$$T(n) = \begin{cases} O(1) & n < 4 \\ 2T(n/2) + O(n) & n \geqslant 4 \end{cases}$$

解此递归方程可得 $T(n) = O(n\log n)$。

这个算法看上去比用排序加扫描的算法复杂,然而它可以推广到以下二维的情形。

设 S 中的点为平面上的点,它们都有两个坐标值 x 和 y。为了将平面上点集 S 线性分割为大小大致相等的两个子集 S_1 和 S_2,我们选取一垂直线 $l: x = m$ 来作为分割直线。其中,m 为 S 中各点 x 坐标的中位数。由此将 S 分割为 $S_1 = \{p \in S \mid x(p) \leqslant m\}$ 和 $S_2 = \{p \in S \mid x(p) > m\}$。从而使 S_1 和 S_2 分别位于直线 l 的左侧和右侧,且 $S = S_1 \cup S_2$。由于 m 是 S 中各点 x 坐标值的中位数,因此 S_1 和 S_2 中的点数大致相等。

递归地在 S_1 和 S_2 上解最接近点对问题,我们分别得到 S_1 和 S_2 中的最小距离 d_1 和 d_2。现设 $d = \min\{d_1, d_2\}$。若 S 的最接近点对(p,q)之间的距离小于 d,则 p 和 q 必分属于 S_1 和 S_2。不妨设 $p \in S_1, q \in S_2$。那么 p 和 q 距直线 l 的距离均小于 d。因此,若我们用 P_1 和 P_2 分别表示直线 l 的左边和右边的宽为 d 的两个垂直长条区域,则 $p \in P_1$ 且 $q \in P_2$,如图 2-9 所示。

在一维情形下,距分割点距离为 d 的两个区间$(m - d, m]$和$(m, m + d]$中最多各有 S 中一个点。因而这两点成为惟一的未检查过的最接近点对候选者。二维的情形则要复杂些,此时,P_1 中所有点与 P_2 中所有点构成的点对均为最接近点对的候选者。在最坏情况下有 $n^2/4$ 对这样的候选者。但是 P_1 和 P_2 中的点具有以下的稀疏性质,它使我们不必检查所有这 $n^2/4$ 个候选者。考虑 P_1 中任意一点 p,它若与 P_2 中的点 q 构成最接近点对的候选者,则必有 distance$(p,q) < d$。满足这个条件的 P_2 中的点有多少个呢?容易看出这样的点一定落在一个 $d \times 2d$ 的矩形 R 中,如图 2-10 所示。

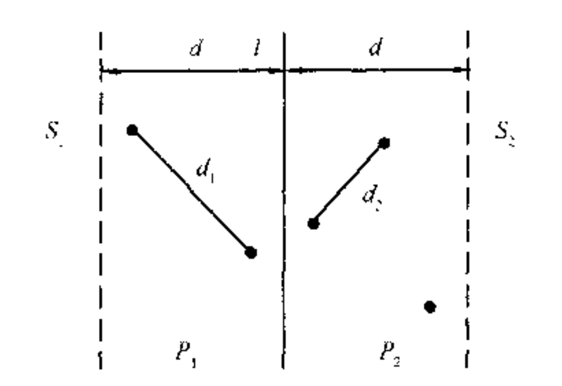


图 2-9 距直线 l 的距离小于 d 的所有点

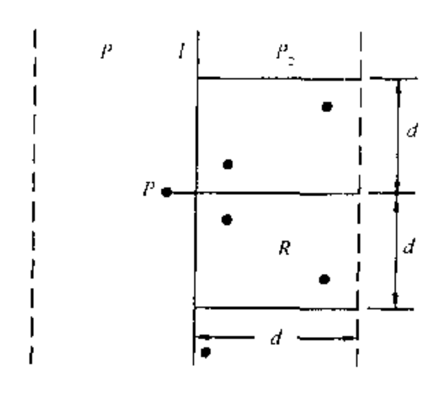


图 2-10 包含点 q 的 $d\times 2d$ 矩形 R

由 d 的意义可知，P_2 中任何两个 S 中的点的距离都不小于 d。由此可以推出矩形 R 中最多只有 6 个 S 中的点。事实上，我们可以将矩形 R 的长为 $2d$ 的边 3 等分，将它的长为 d 的边 2 等分，由此导出 6 个$(d/2)\times(2d/3)$ 的矩形。如图 2-11(a) 所示。

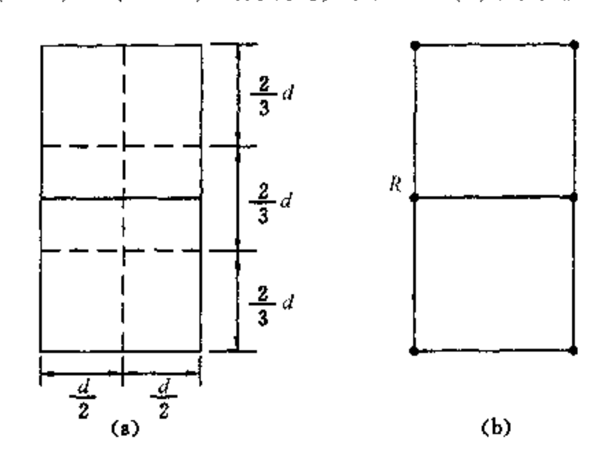


图 2-11 矩形 R 中点的稀疏性

若矩形 R 中有多于 6 个 S 中的点，则由鸽舍原理易知至少有一个$(d/2)\times(2d/3)$ 的小矩形中有 2 个以上 S 中的点。设 u,v 是这样 2 个点，它们位于同一小矩形中，则

$$(x(u)-x(v))^2+(y(u)-y(v))^2\leqslant(d/2)^2+(2d/3)^2=\frac{25}{36}d^2$$

因此，distance$(u,v)\leqslant 5d/6<d$。这与 d 的意义相矛盾。也就是说矩形 R 中最多只有 6 个 S 中的点。图 2-11(b) 是矩形 R 中恰有 6 个 S 中点的极端情形。由于这种稀疏性质，对于 P_1 中任一点 p，P_2 中最多只有 6 个点与它构成最接近点对的候选者。因此，在分治法的合并步骤中，我们最多只需要检查 $6\times n/2=3n$ 个候选者，而不是 $n^2/4$ 个候选者。这是否就意味着我们可以在 $O(n)$ 时间内完成分治法的合并步骤呢？现在还不能作出这个结论。因为我们只知道对于 P_1 中每个 S_1 中的点最多只需要检查 S_2 中 6 个点，但是我们并不确切地知道要检查哪 6 个点。为解决这一问题，我们可以将 p 和 P_2 中所有 S_2 的点投影到垂直线 l 上。由于能与 p 点一起构成最接近点对候选者的 S_2 中点一定在矩形 R 中，所以它们在直线 l 上的投影点距 p 在 l 上投影点的距离小于 d。由上面的分析可知，这种投影点最多只有 6 个。因此，若将 P_1 和 P_2 中所有 S 中点按其 y 坐标排好序，则对 P_1 中所有点，对排好序的点列作一次扫描，就可以找出所有最接近点对的候选者。对 P_1 中每一点最多只要检查 P_2 中排好序的相继 6 个点。

至此，我们给出用分治法求二维点集最接近点对的算法 Cpair2 如下：

```
bool Cpair2(S,d)
{
      n = |S|;
      if (n < 2) { d = ∞;
                  return false;
                  }
1.    m = S中各点x间坐标的中位数;
      构造S1和S2;
      //S1 = {p∈S|x(p) <= m}, S2 = {p∈S|x(p) > m}
2.    Cpair2(S1,d1);
      Cpair2(S2,d2);
3.    dm = min(d1,d2);
4.    设P1是S1中距垂直分割线l的距离在dm之内的所有点组成的集合;
      P2是S2中距分割线l的距离在dm之内所有点组成的集合;
      将P1和P2中点依其y坐标值排序;
      并设X和Y是相应的已排好序的点列;
5.    通过扫描X以及对于X中每个点检查Y中与其距离在dm之内的所有点(最多6
      个)可以完成合并;
      当X中的扫描指针逐次向上移动时,Y中的扫描指针可在宽为2dm的一个区间内
      移动;
      设dl是按这种扫描方式找到的点对间的最小距离;
6.    d = min(dm,dl);
      return true;
}
```

下面分析算法Cpair2的计算复杂性。设对于n个点的平面点集S,算法耗时$T(n)$。算法的第1步和第5步用了$O(n)$时间。第3步和第6步用了常数时间。第2步用了$2T(n/2)$时间。若在每次执行第4步时进行排序,则在最坏情况下第4步要用$O(n\log n)$时间。这不符合我们的要求。因此,在这里我们要作一个技术处理。我们采用设计算法时常用的预排序技术,在使用分治法之前,预先将S中n个点依其y坐标值排好序,设排好序的点列为$P*$。在执行分治法的第4步时,只要对$P*$作一次线性扫描,即可抽取出我们所需要的排好序的点列X和Y。然后,在第5步中再对X作一次线性扫描,即可求得dl。因此,第4步和第5步的两遍扫描合在一起只要用$O(n)$时间。这样,经过预排序处理后算法Cpair2所需的计算时间$T(n)$满足递归方程

$$T(n) = \begin{cases} O(1) & n < 4 \\ 2T(n/2) + O(n) & n \geq 4 \end{cases}$$

由此易知,$T(n) = O(n\log n)$。预排序所需的计算时间显然为$O(n\log n)$。因此,整个算法所需的计算时间为$O(n\log n)$。在渐近的意义下,此算法已是最优算法。

在具体实现算法Cpair2时,我们分别用类PointX和PointY表示依x坐标和依y坐标排序的点。

```
class PointX {
   public:
      int operator <= (PointX a) const
      {return (x <= a.x);}
```

```
    private:
        int ID;      // 点编号
        float x, y;  // 点坐标
};

class PointY {
    public:
        int operator <= (PointY a) const
        {return (y <= a.y);}
    private:
        int p;      // 同一点在数组 X 中的坐标
        float x, y;  // 点坐标
};
```

平面上任意两点 u 和 v 之间的距离可计算如下:

```
template < class Type >
inline float distance(const Type& u, const Type& v)
{
    float dx = u.x - v.x;
    float dy = u.y - v.y;
    return sqrt(dx * dx + dy * dy);
}
```

在算法 Cpair2 中,用数组 X 存储输入的点集。在算法的预处理阶段,将数组 X 中的点依 x 坐标和依 y 坐标排序,排好序的点集分别存储在数组 X 和数组 Y 中。经过预排序后,在算法的分割阶段,将子数组 X[$l:r$] 均匀地划分成两个不相交的子集的任务就可以在 $O(1)$ 时间内完成。事实上,我们只要取 $m = (l + r)/2$,则 X[$l:m$] 和 X[$m+1:r$] 就是满足要求的分割。依 y 坐标排好序的数组 Y 用于在算法的合并步中快速检查 d 矩形条内最接近点对的候选者。

```
bool Cpair2(PointX X[], int n, PointX& a,
                         PointX& b, float& d)
{
    if (n < 2) return false;
    MergeSort(X,n);
    PointY * Y = new PointY [n];
    for (int i = 0; i < n; i++) {
        // 将数组 X 中的点复制到数组 Y 中
        Y[i].p = i;
        Y[i].x = X[i].x;
        Y[i].y = X[i].y;
        }
    MergeSort(Y,n);
    PointY * Z = new PointY [n];
    closest(X,Y,Z,0,n - 1,a,b,d);
    delete [] Y;
    delete [] Z;
```

```
        return true;
    }
```

算法 Cpair2 中,具体计算最接近点对的工作由函数 closest 完成。

```
void closest(PointX X[], PointY Y[], PointY Z[],
        int l, int r, PointX& a, PointX& b, float& d)
{
    if (r - l == 1) {// 2点的情形
        a = X[l];
        b = X[r];
        d = distance(X[l], X[r]);
        return;}
    if (r - l == 2) {// 3点的情形
        float d1 = distance(X[l], X[l + 1]);
        float d2 = distance(X[l + 1], X[r]);
        float d3 = distance(X[l], X[r]);
        if (d1 <= d2 && d1 <= d3) {
            a = X[l];
            b = X[l + 1];
            d = d1;
            return;}
        if (d2 <= d3) {a = X[l + 1];
                       b = X[r];
                       d = d2;}
        else { a = X[l];
               b = X[r];
               d = d3;}
        return;}
    // 多于3点的情形,用分治法
    int m = (l + r)/2;
    int f = l,
        g = m + 1;
    for (int i = l; i <= r; i++)
        if (Y[i].p > m) Z[g++] = Y[i];
        else Z[f++] = Y[i];
    closest(X,Z,Y,l,m,a,b,d);
    float dr;
    PointX ar, br;
    closest(X,Z,Y,m + 1,r,ar,br,dr);
    if (dr < d) {a = ar;
                 b = br;
                 d = dr;}
    Merge(Z,Y,l,m,r);// 重构数组 Y
    // d 矩形条内的点置于 Z 中
    int k = l;
    for (int i = l; i <= r; i++)
```

```
            if (fabs(Y[m].x - Y[i].x) < d) Z[k++] = Y[i];
        // 搜索 Z[l:k - 1]
        for (int i = l; i < k; i++){
            for (int j = i + 1; j < k && Z[j].y - Z[i].y < d;
                        j++){
                float dp = distance(Z[i], Z[j]);
                if (dp < d) {
                        d = dp;
                        a = X[Z[i].p];
                        b = X[Z[j].p];}
                }
            }
    }
```

2.11 循环赛日程表

分治法不仅可以用来设计算法,而且在其他方面也有广泛应用。例如可以用分治思想来设计电路、构造数学证明等。现举一例加以说明。

设有 $n = 2^k$ 个运动员要进行网球循环赛。现要设计一个满足以下要求的比赛日程表:

(1) 每个选手必须与其他 $n - 1$ 个选手各赛一次。

(2) 每个选手一天只能赛一次。

(3) 循环赛一共进行 $n - 1$ 天。

按此要求可将比赛日程表设计成有 n 行和 $n - 1$ 列的一个表。在表中第 i 行和第 j 列处填入第 i 个选手在第 j 天所遇到的选手。

按分治策略,我们可以将所有选手对分为两组,n 个选手的比赛日程表就可以通过为 $n/2$ 个选手设计的比赛日程表来决定。递归地用这种一分为二的策略对选手进行分割,直到只剩下 2 个选手时,比赛日程表的制定就变得很简单。这时只要让这 2 个选手进行比赛就可以了。

图 2-12 所列出的正方形表是 8 个选手的比赛日程表。其中左上角与左下角的两小块分别为选手 1 至选手 4 和选手 5 至选手 8 前 3 天的比赛日程。据此,将左上角小块中的所有数字按其相对位置抄到右下角,将左下角小块中的所有数字按其相对位置抄到右上角,这样我们就分别安排好了选手 1 至选手 4 和选手 5 至选手 8 在后 4 天的比赛日程。依此思想容易将这个比赛日程表推广到具有任意多个选手的情形。

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

图 2-12 8 个选手的比赛日程表

在一般情况下，算法可描述如下：

```
void Table(int k, int * * a)
{
    int n = 1;
    for (int i = 1; i < = k; i ++) n * = 2;
    for (int i = 1; i < = n; i ++) a[1][i] = i;
    int m = 1;
    for (int s = 1; s < = k; s ++) {
      n/ = 2;
      for (int t = 1; t < = n; t ++)
        for (int i = m + 1; i < = 2 * m; i ++)
          for (int j = m + 1; j < = 2 * m; j ++) {
            a[i][j + (t - 1) * m * 2] = a[i - m][j + (t - 1) * m * 2 - m];
            a[i][j + (t - 1) * m * 2 - m] = a[i - m][j + (t - 1) * m * 2];}
      m * = 2;
      }
}
```

习题 2

2-1 证明 Hanoi 塔问题的递归算法与非递归算法实际上是一回事。

2-2 下面的 7 个算法与本章中的二分搜索算法 BinarySearch 略有不同。请判断这 7 个算法的正确性。如果算法不正确，请说明产生错误的原因。如果算法正确，请给出算法的正确性证明。

```
template < class Type >
int BinarySearch1(Type a[], const Type& x, int n)
{
    int left = 0; int right = n - 1;
    while (left < = right) {
        int middle = (left + right)/2;
        if (x = = a[middle]) return middle;
        if (x > a[middle]) left = middle;
        else right = middle;
        }
    return - 1;
}
```

```
template < class Type >
int BinarySearch2(Type a[], const Type& x, int n)
{
    int left = 0; int right = n - 1;
    while (left < right - 1) {
        int middle = (left + right)/2;
```

```
            if (x < a[middle]) right = middle;
            else left = middle;
            }
        if (x == a[left]) return left;
        else return -1;
    }
```

```
template <class Type>
int BinarySearch3(Type a[], const Type& x, int n)
{
    int left = 0; int right = n - 1;
    while (left + 1 != right) {
        int middle = (left + right)/2;
        if (x >= a[middle]) left = middle;
        else right = middle;
        }
    if (x == a[left]) return left;
    else return -1;
}
```

```
template <class Type>
int BinarySearch4(Type a[], const Type& x, int n)
{
    if (n > 0 && x >= a[0]) {
    int left = 0; int right = n - 1;
    while (left < right) {
        int middle = (left + right)/2;
        if (x < a[middle]) right = middle - 1;
        else left = middle ;
        }
    if (x == a[left]) return left;
    }
    return -1;
}
```

```
template <class Type>
int BinarySearch5(Type a[], const Type& x, int n)
{
    if (n > 0 && x >= a[0]) {
    int left = 0; int right = n - 1;
    while (left < right) {
        int middle = (left + right + 1)/2;
        if (x < a[middle]) right = middle - 1;
        else left = middle ;
```

```
    }
    if (x == a[left]) return left;
    }
    return -1;
}
```

```
template <class Type>
int BinarySearch6(Type a[], const Type& x, int n)
{
    if (n > 0 && x >= a[0]) {
    int left = 0; int right = n - 1;
    while (left < right) {
        int middle = (left + right + 1)/2;
        if (x < a[middle]) right = middle - 1;
        else left = middle + 1 ;
        }
    if (x == a[left]) return left;
    }
    return -1;
}
```

```
template <class Type>
int BinarySearch7(Type a[], const Type& x, int n)
{
    if (n > 0 && x >= a[0]) {
    int left = 0; int right = n - 1;
    while (left < right) {
      int middle = (left + right + 1)/2;
      if (x < a[middle]) right = middle;
      else left = middle ;
      }
    if (x == a[left]) return left;
    }
    return -1;
}
```

2-3　设a[0: $n-1$]是一个已排好序的数组。请改写二分搜索算法,使得当搜索元素 x 不在数组中时,返回小于 x 的最大元素位置 i 和大于 x 的最小元素位置 j。当搜索元素在数组中时,i 和 j 相同,均为 x 在数组中的位置。

2-4　给定两个整数 u 和 v,它们分别有 m 和 n 位数字,且 $m \leqslant n$。用通常的乘法求 uv 的值需要 $O(mn)$ 时间。我们可以将 u 和 v 均看作是有 n 位数字的大整数,用本章介绍的分治法,在 $O(n^{\log 3})$ 时间内计算 uv 的值。当 m 比 n 小得多时,用这种方法就显得效率不够高。试设计一个算法,在上述情况下用 $O(nm^{\log(3/2)})$ 时间求出 uv 的值。

2-5　我们在用分治法求两个 n 位大整数 u 和 v 的乘积时,将 u 和 v 都分割为长度为 $n/3$

位的3段。证明可以用5次 $n/3$ 位整数的乘法求得 uv 的值。按此思想设计一个求两个大整数乘积的分治算法，并分析算法的计算复杂性。（提示：n 位的大整数除以一个常数 k 可以在 $\theta(n)$ 时间内完成。符号 θ 所隐含的常数可能依赖于 k。）

2-6　对任何非零偶数 n，总可以找到一个奇数 m 和一个正整数 k，使得 $n = m2^k$。为了求出两个 n 阶矩阵的乘积，可以把一个 n 阶矩阵分成 $m \times m$ 个子矩阵，每个子矩阵有 $2^k \times 2^k$ 个元素。当需要求 $2^k \times 2^k$ 的子矩阵的积时，使用Strassen算法。设计一个传统方法与Strassen算法相结合的矩阵相乘算法，对任何偶数 n，都可以求出两个 n 阶矩阵的乘积。并分析算法的计算时间复杂性。

2-7　设 $P(x) = a_0 + a_1x + \cdots + a_dx^d$ 是一个 d 次多项式。假设已有一算法能在 $O(i)$ 时间内计算一个 i 次多项式与一个一次多项式的乘积，以及一个算法能在 $O(i\log i)$ 时间内计算两个 i 次多项式的乘积。对于任意给定的 d 个整数 $n_1, n_2, \cdots, n_d$，用分治法设计一个有效算法，计算出满足 $P(n_1) = P(n_2) = \cdots = P(n_d) = 0$ 且最高次项系数为1的 d 次多项式 $P(x)$，并分析算法的效率。

2-8　设 n 个不同的整数排好序后存于 T[0: n - 1] 中。若存在一个下标 i，$0 \leq i < n$，使得 T[i] = i，设计一个有效算法找到这个下标。要求算法在最坏情况下的计算时间为 $O(\log n)$。

2-9　设T[0: n - 1] 是 n 个元素的一个数组。对任一元素 x，设 $S(x) = \{i \mid T[i] = x\}$。当 $|S(x)| > n/2$ 时，称 x 为 T 的主元素。设计一个线性时间算法，确定 T[0: n - 1] 是否有一个主元素。

2-10　若在习题2-9 中，数组 T 中元素不存在序关系，只能测试任意两个元素是否相等，试设计一个有效算法确定 T 是否有一主元素。算法的计算复杂性应为 $O(n\log n)$。更进一步，能找到一个线性时间算法吗？

2-11　设 a[0: n - 1] 是一个有 n 个元素的数组，$k(0 \leq k \leq n - 1)$ 是一个非负整数。试设计一个算法将子数组 a[0: k] 与 a[k + 1: n - 1] 换位。要求算法在最坏情况下耗时 $O(n)$，且只用到 $O(1)$ 的辅助空间。

2-12　设子数组 a[0: k] 和 a[k + 1: n - 1] 已排好序 $(0 \leq k \leq n - 1)$。试设计一个合并这两个子数组为排好序的数组 a[0: n - 1] 的算法。要求算法在最坏情况下所用的计算时间为 $O(n)$，且只用到 $O(1)$ 的辅助空间。

2-13　如果我们在合并排序算法的分割步骤中，将数组 a[0: n - 1] 划分为 $\lfloor \sqrt{n} \rfloor$ 个子数组，每个子数组中有 $O(\sqrt{n})$ 个元素。然后递归地对分割后的子数组进行排序，最后将所得到的 $\lfloor \sqrt{n} \rfloor$ 个排好序的子数组合并成所要求的排好序的数组 a[0: n - 1]。设计一个实现上述策略的合并排序算法，并分析算法的计算复杂性。

2-14　对所给元素存储于数组中和存储于链表中两种情形，写出自然合并排序算法。

2-15　给定数组 a[0: n - 1]，试设计一个算法，在最坏情况下用 $\lceil 3n/2 - 2 \rceil$ 次比较找出 a[0: n - 1] 中元素的最大值和最小值。

2-16　给定数组 a[0: n - 1]，试设计一个算法，在最坏情况下用 $n + \lceil \log n \rceil - 2$ 次比较找出 a[0: n - 1] 中元素的最大值和次大值。

2-17　设 $S_1, S_2, \cdots, S_k$ 是整数集合，其中每个集合 $S_i(1 \leq i \leq k)$ 中整数取值范围是1到

n,且$\sum_{i=1}^{k}|S_i| = n$,试设计一个算法在 $O(n)$ 时间内将 $S_1,S_2,\cdots,S_k$ 分别排序。

2-18　试证明,在最坏情况下,求 n 个元素组成的集合 S 中的第 k 小元素至少需要 $n + \min(k, n-k+1) - 2$ 次比较。

2-19　如何修改 QuickSort 才能使其将输入元素按非增序排序?

2-20　对一个随机化算法,为什么我们只分析其平均情况下的性能,而不分析其最坏情况下的性能?

2-21　在执行 RandomizedQuicksort 时,在最坏情况下,调用 Random 多少次?在最好情况下又怎样?

2-22　试设计一个 $O(n)$ 时间算法,使之能产生数组 a[0:$n-1$] 元素的一个随机排列。

2-23　试用 while 循环消去算法 QuickSort 中的尾递归,并比较消去尾递归前后算法的效率。

2-24　试用栈来模拟递归,消去算法 QuickSort 中的递归。并证明所需的栈空间为 $O(\log n)$。

2-25　在算法 Select 中,输入元素被划分为5个一组,如果将它们划分为7个一组,该算法仍然是线性时间算法吗?划分成 3 个一组又怎样?

2-26　试说明如何修改快速排序算法,使它在最坏情况下的计算时间为 $O(n\log n)$。

2-27　给定一个由 n 个互不相同的数组成的集合 S,及一个正整数 $k \leqslant n$,试设计一个 $O(n)$ 时间算法找出 S 中最接近 S 的中位数的 k 个数。

2-28　设 X[0:$n-1$] 和 Y[0:$n-1$] 为两个数组,每个数组中含有 n 个已排好序的数。试设计一个 $O(\log n)$ 时间的算法,找出 X 和 Y 的 $2n$ 个数的中位数。

2-29　考察如图 2-13 所示的有两个输入端和两个输出端的二位置开关。当开关处于位置 1 时,输入 1 和 2 分别产生输出 1 和 2;当开关处于位置 2 时,输入 1 和 2 分别产生输出 2 和 1。使用这种开关设计一个有 n 个输入端和 n 个输出端的开关网络,实现将输入的 n 个数值以它们的 n!种不同排列的任何一种排列输出(通过开关位置的适当选择)。要求网络中使用的开关个数为 $O(n\log n)$。

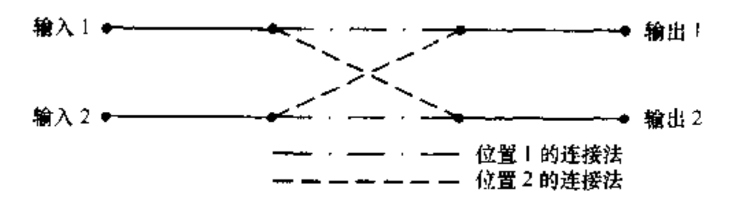


图 2-13　二位置开关

2-30　某石油公司计划建造一条由东向西的主输油管道。该管道要穿过一个有 n 口油井的油田。从每口油井都要有一条输油管道沿最短路经(或南或北)与主管道相连。如果给定 n 口油井的位置,即它们的 x 坐标和 y 坐标,应如何确定主管道的最优位置,即使各油井到主管道之间的输油管道长度总和最小的位置?证明可在线性时间内确定主管道的最优位置。

2-31　在一个由元素组成的表中,出现次数最多的元素称为众数。试写一个寻找众数的算法,并分析其计算复杂性。

2-32　对于 n 个带有正权 $w_1, w_2, \cdots, w_n$，且 $\sum_{i=1}^{n} w_i = 1$ 的互不相同的元素 $x_1, x_2, \cdots, x_n$，其带权中位数 x_k 满足：

$$\begin{cases} \sum_{x_i < x_k} w_i \leqslant \frac{1}{2} \\ \sum_{x_i > x_k} w \leqslant \frac{1}{2} \end{cases}$$

(1) 试证明 $x_1, x_2, \cdots, x_n$ 的不带权中位数是带权 $w_i = 1/n, i = 1,2,\cdots,n$ 的带权中位数；

(2) 说明如何通过排序，在最坏情况下用 $O(n\log n)$ 时间求出 n 个元素的带权中位数；

(3) 说明如何利用一个线性时间选择算法(如 Select)，在最坏情况下用 $O(n)$ 时间求出 n 个元素的带权中位数；

(4) 邮局位置问题定义为：已知 n 个点 $p_1, p_2, \cdots, p_n$ 以及与它们相联系的权 $w_1, w_2, \cdots, w_n$，要求确定一点 p(p 不一定是 n 个输入点之一)，使和式 $\sum_{i=1}^{n} w_i d(p, p_i)$ 达到最小，其中，$d(a,b)$ 表示 a 与 b 之间的距离。试论证带权中位数是一维邮局问题的最优解。此时 $d(a,b) = |a - b|$。

(5) 在二维的情形如何找最优解？

2-33　考虑国际象棋棋盘上某个位置的一只马，它是否可能只走 63 步，正好走过除起点外的其他 63 个位置各一次？如果有一种这样的走法，则称所走的这条路线为一条马的周游路线。试设计一个分治算法找出这样的一条马的周游路线。

2-34　Gray 码是一个长度为 2^n 的序列。序列中无相同元素，每个元素都是长度为 n 位的串，相邻元素恰好只有一位不同。用分治策略设计一个算法对任意的 n 构造相应的 Gray 码。

2-35　设有 n 个运动员要进行网球循环赛。设计一个满足以下要求的比赛日程表：

(1) 每个选手必须与其他 $n - 1$ 个选手各赛一次。

(2) 每个选手一天只能赛一次。

(3) 当 n 是偶数时，循环赛进行 $n - 1$ 天。当 n 是奇数时，循环赛进行 n 天。

第3章　动态规划

学习要点

· 理解动态规划算法的概念
· 掌握动态规划算法的基本要素：
(1) 最优子结构性质
(2) 重叠子问题性质
· 掌握设计动态规划算法的步骤：
(1) 找出最优解的性质，并刻画其结构特征
(2) 递归地定义最优值
(3) 以自底向上的方式计算最优值
(4) 根据计算最优值时得到的信息构造最优解
· 通过下面的应用范例学习动态规划算法设计策略：
(1) 矩阵连乘问题
(2) 最长公共子序列
(3) 最大子段和
(4) 凸多边形最优三角剖分
(5) 多边形游戏
(6) 图像压缩
(7) 电路布线
(8) 流水作业调度
(9) 背包问题
(10) 最优二叉搜索树

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划法求解的问题，经分解得到的子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，以至于最后解决原问题需要耗费指数时间。然而，不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，从而得到多项式时间的算法。为了达到这个目的，我们可以用一个表来记录所有已解决的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表格式。

动态规划算法通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值（最大值或最小值）的那个解。设计一个动态规划算法，通常可按以下几个步骤进行：

(1) 找出最优解的性质，并刻画其结构特征。

(2) 递归地定义最优值。

(3) 以自底向上的方式计算出最优值。

(4) 根据计算最优值时得到的信息,构造一个最优解。

步骤(1) ~ (3)是动态规划算法的基本步骤。在只需要求出最优值的情形,步骤(4)可以省去。若需要求出问题的一个最优解,则必须执行步骤(4)。此时,在步骤(3)中计算最优值时,通常需记录更多的信息,以便在步骤(4)中,根据所记录的信息,快速构造出一个最优解。

下面我们以具体的例子来说明如何运用动态规划算法求解问题,并分析可用动态规划算法求解的问题所应具备的一般特征。

3.1 矩阵连乘问题

给定 n 个矩阵 $\{A_1, A_2, \cdots, A_n\}$,其中,$A_i$ 与 A_{i+1} 是可乘的,$i = 1,2,\cdots,n-1$。我们要计算出这 n 个矩阵的连乘积 $A_1A_2,\cdots,A_n$。

由于矩阵乘法满足结合律,故计算矩阵的连乘积可以有许多不同的计算次序。这种计算次序可以用加括号的方式来确定。若一个矩阵连乘积的计算次序完全确定,也就是说该连乘积已完全加括号,则我们可依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积。完全加括号的矩阵连乘积可递归地定义为:

(1) 单个矩阵是完全加括号的;

(2) 矩阵连乘积 A 是完全加括号的,则 A 可表示为2个完全加括号的矩阵连乘积 B 和 C 的乘积并加括号,即 $A = (BC)$。

例如,矩阵连乘积 $A_1A_2A_3A_4$ 可以有以下5种不同的完全加括号方式:

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

每一种完全加括号方式对应于一种矩阵连乘积的计算次序,而这种计算次序与计算矩阵连乘积的计算量有着密切的关系。

首先我们来考虑计算2个矩阵乘积所需的计算量。

计算2个矩阵乘积的标准算法如下,其中,ra,ca和rb,cb分别表示矩阵 A 和 B 的行数和列数。

```
void matrixMultiply(int **a, int **b, int **c, int ra, int ca, int rb, int cb)
{
    if (ca != rb) error ("矩阵不可乘");
    for (int i = 0; i < ra; i++)
     for (int j = 0; j < cb; j++) {
        int sum = a[i][0] * b[0][j];
        for (int k = 1; k < ca; k++)
           sum += a[i][k] * b[k][j];
        c[i][j] = sum;
        }
}
```

矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 是可乘的条件是矩阵 $\boldsymbol{A}$ 的列数等于矩阵 $\boldsymbol{B}$ 的行数。若 $\boldsymbol{A}$ 是一个 $p \times q$ 矩阵，$\boldsymbol{B}$ 是一个 $q \times r$ 矩阵，则其乘积 $\boldsymbol{C} = \boldsymbol{AB}$ 是一个 $p \times r$ 矩阵。在上述计算 $\boldsymbol{C}$ 的标准算法中，主要计算量在三重循环，总共需要 pqr 次数乘。

为了说明在计算矩阵连乘积时，加括号方式对整个计算量的影响，我们来看一个计算 3 个矩阵 $\{\boldsymbol{A}_1, \boldsymbol{A}_2, \boldsymbol{A}_3\}$ 的连乘积的例子。设这 3 个矩阵的维数分别为 10×100, 100×5, 和 5×50。若按第一种加括号方式 $((\boldsymbol{A}_1\boldsymbol{A}_2)\boldsymbol{A}_3)$ 来计算，则计算 3 个矩阵连乘积需要的数乘次数为 $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7\ 500$。若按第二种加括号方式 $(\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))$ 来计算 3 个矩阵连乘积总共需要 $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75\ 000$ 次数乘。第二种加括号方式的计算量是第一种加括号方式计算量的 10 倍。由此可见，在计算矩阵连乘积时，加括号方式，即计算次序对计算量有很大影响。于是，人们自然会提出矩阵连乘积的最优计算次序问题，即对于给定的 n 个矩阵 $\{\boldsymbol{A}_1, \boldsymbol{A}_2, \cdots, \boldsymbol{A}_n\}$（其中，矩阵 $\boldsymbol{A}_i$ 的维数为 $p_{i-1} \times p_i$, $i = 1, 2, \cdots, n$），如何确定计算矩阵连乘积 $\boldsymbol{A}_1\boldsymbol{A}_2, \cdots, \boldsymbol{A}_n$ 的一个计算次序（完全加括号方式），使得依此次序计算矩阵连乘积需要的数乘次数最少。

穷举搜索法是最容易想到的解法。算法列举出所有可能的计算次序，并计算出每一种计算次序相应需要的数乘次数，由此找出一种所需数乘次数最少的计算次序。然而，这样做计算量太大。事实上，对于 n 个矩阵的连乘积，设有 $P(n)$ 个不同的计算次序。由于我们可以先在第 k 个和第 $k+1$ 个矩阵之间将原矩阵序列分为两个矩阵子序列，$k = 1, 2, \cdots, n-1$；然后分别对这两个矩阵子序列完全加括号；最后对所得的结果加括号，得到原矩阵序列的一种完全加括号方式。由此，可以得到关于 $P(n)$ 的递归式如下：

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$$

解此递归方程可得，$P(n)$ 实际上是 Catalan 数，即 $P(n) = C(n-1)$，其中，

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n / n^{3/2})$$

也就是说，$P(n)$ 是随 n 的增长呈指数增长的。因此，穷举搜索法不是一个有效算法。

下面我们考虑用动态规划法解矩阵连乘积的最优计算次序问题。如前所述，我们按以下几个步骤来进行。

1. 分析最优解的结构

设计求解具体问题的动态规划算法的第 1 步是刻画该问题的最优解结构特征。为方便起见，将矩阵连乘积 $\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_j$ 简记为 $\boldsymbol{A}[i:j]$。我们来看计算 $\boldsymbol{A}[1:n]$ 的一个最优次序。设这个计算次序在矩阵 $\boldsymbol{A}_k$ 和 $\boldsymbol{A}_{k+1}$ 之间将矩阵链断开，$1 \leqslant k < n$，则完全加括号方式为 $((\boldsymbol{A}_1\cdots\boldsymbol{A}_k)(\boldsymbol{A}_{k+1}\cdots\boldsymbol{A}_n))$。依此次序，我们先分别计算 $\boldsymbol{A}[1:k]$ 和 $\boldsymbol{A}[k+1:n]$，然后将计算结果相乘得到 $\boldsymbol{A}[1:n]$，总计算量为 $\boldsymbol{A}[1:k]$ 的计算量加上 $\boldsymbol{A}[k+1:n]$ 的计算量，再加上 $\boldsymbol{A}[1:k]$ 和 $\boldsymbol{A}[k+1:n]$ 相乘的计算量。

这个问题的一个关键特征是：计算 $\boldsymbol{A}[1:n]$ 的一个最优次序所包含的计算矩阵子链 $\boldsymbol{A}[1:k]$ 和 $\boldsymbol{A}[k+1:n]$ 的次序也是最优的。事实上，若有一个计算 $\boldsymbol{A}[1:k]$ 的次序需要的计算量更少，则用此次序替换原来计算 $\boldsymbol{A}[1:k]$ 的次序，得到的计算 $\boldsymbol{A}[1:n]$ 的次序需要的计算量将比最优次序所需计算量更少，这是一个矛盾。同理可知，计算 $\boldsymbol{A}[1:n]$ 的一个最优次序所包含的

计算矩阵子链 $A[k+1:n]$ 的次序也是最优的。

因此，矩阵连乘积计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。一个问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

2. 建立递归关系

设计一个动态规划算法的第 2 步是递归地定义最优值。对于矩阵连乘积的最优计算次序问题，设计算 $A[i:j]$，$1 \leqslant i \leqslant j \leqslant n$，所需的最少数乘次数为 $m[i][j]$，则原问题的最优值为 $m[1][n]$。

当 $i=j$ 时，$A[i:j]=A_i$ 为单一矩阵，无需计算，因此 $m[i][i]=0$，$i=1,2,\cdots,n$。

当 $i<j$ 时，可利用最优子结构性质来计算 $m[i][j]$。事实上，若计算 $A[i:j]$ 的最优次序在 A_k 和 A_{k+1} 之间断开，$i \leqslant k < j$，则 $m[i][j]=m[i][k]+m[k+1][j]+p_{i-1}p_kp_j$。由于在计算时我们并不知道断开点 k 的位置，所以 k 还未定。不过 k 的位置只有 $j-i$ 个可能，即 $k \in \{i,i+1,\cdots,j-1\}$。因此，$k$ 是这 $j-i$ 个位置中使计算量达到最小的那个位置。从而 $m[i][j]$ 可以递归地定义为

$$m[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\leqslant k<j}\{m[i][k]+m[k+1][j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

$m[i][j]$ 给出了最优值，即计算 $A[i:j]$ 所需的最少数乘次数。同时还确定了计算 $A[i:j]$ 的最优次序中的断开位置 k，也就是说，对于这个 k 有

$$m[i][j]=m[i][k]+m[k+1][j]+p_{i-1}p_kp_j$$

若将对应于 $m[i][j]$ 的断开位置 k 记为 $s[i][j]$，在计算出最优值 $m[i][j]$ 后，可递归地由 $s[i][j]$ 构造出相应的最优解。

3. 计算最优值

根据计算 m[i][j] 的递归式，容易写一个递归算法来计算 m[1][n]。稍后我们将看到，简单地递归计算将耗费指数计算时间。然而，我们注意到，在递归计算过程中，不同的子问题个数只有 $\theta(n^2)$ 个。事实上，对于 $1 \leqslant i \leqslant j \leqslant n$ 不同的有序对 (i,j) 对应于不同的子问题。因此，不同子问题的个数最多只有 $\binom{n}{2}+n=\theta(n^2)$ 个。由此可见，在递归计算时，许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。

用动态规划算法解此问题，可依据其递归式以自底向上的方式进行计算。在计算过程中，保存已解决的子问题答案。每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算，最终得到多项式时间的算法。下面所给出的计算 m[i][j] 的动态规划算法 MatrixChain 中，输入参数 $\{p_0,p_1,\cdots,p_n\}$ 存储于数组 p 中。算法除了输出最优值数组 m 外还输出记录最优断开位置的数组 s。

```
void MatrixChain(int * p, int n, int * * m, int * * s)
{
  for (int i = 1; i < = n; i ++) m[i][i] = 0;
  for (int r = 2; r < = n; r ++)
    for (int i = 1; i < = n - r + 1; i ++) {
      int j = i + r - 1;
```

```
        m[i][j] = m[i+1][j] + p[i-1] * p[i] * p[j];
      s[i][j] = i;
        for (int k = i+1; k < j; k++) {
          int t = m[i][k] + m[k+1][j]
                  + p[i-1] * p[k] * p[j];
          if (t < m[i][j]) {
              m[i][j] = t;
            s[i][j] = k;}
          }
        }
    }
```

算法 MatrixChain 首先计算出 m[i][i] = 0, $i = 1,2,\cdots,n$,然后,再根据递归式,按矩阵链长递增的方式依次计算 m[i][$i+1$], $i = 1,2,\cdots,n-1$,(矩阵链长度为 2);m[i][$i+2$], $i = 1,2,\cdots,n-2$,(矩阵链长度为 3);…。在计算 m[i][j] 时,只用到已计算出的 m[i][k] 和 m[$k+1$][j]。

例:设要计算矩阵连乘积 $A_1A_2A_3A_4A_5A_6$,其中各矩阵的维数分别为:

A_1	A_2	A_3	A_4	A_5	A_6
30×35	35×15	15×5	5×10	10×20	20×25

动态规划算法 MatrixChain 计算 m[i][j] 先后次序如图 3-1(a) 所示;计算结果 m[i][j] 和 s[i][j], $1 \leqslant i \leqslant j \leqslant n$,分别如图 3-1(b) 和(c) 所示。

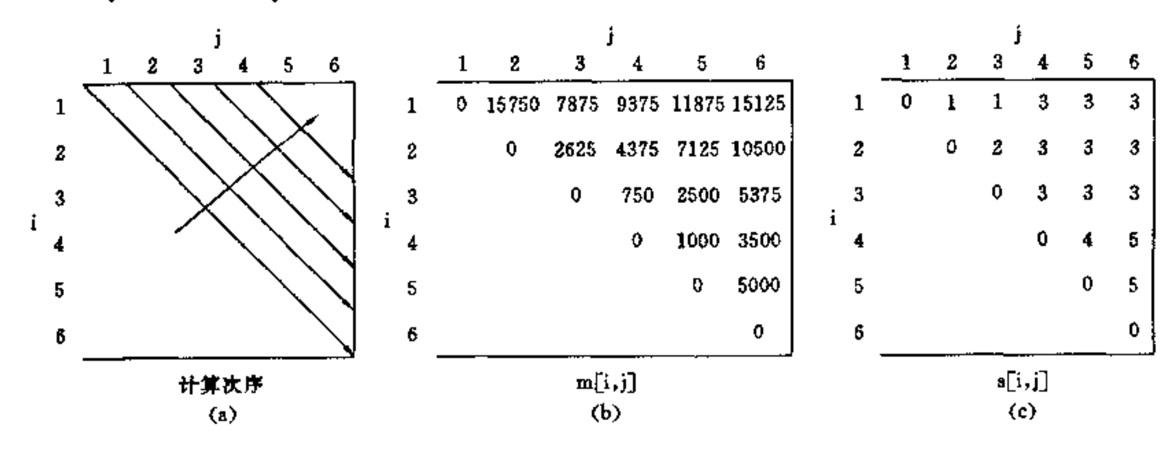


图 3-1 计算 m[i][j] 的次序

例如,在计算 m[2][5] 时,依递归式有

$$
\mathrm{m}[2][5] = \min\begin{cases} \mathrm{m}[2][2] + \mathrm{m}[3][5] + p_1p_2p_5 = 0 + 2500 + 35 \times 15 \times 20 = 13\,000 \\ \mathrm{m}[2][3] + \mathrm{m}[4][5] + p_1p_3p_5 = 2625 + 1000 + 35 \times 5 \times 20 = 7\,125 \\ \mathrm{m}[2][4] + \mathrm{m}[5][5] + p_1p_4p_5 = 4375 + 0 + 35 \times 10 \times 20 = 11\,375 \end{cases}
$$

$$
= 7\,125
$$

且 $k = 3$,因此,s[2][5] = 3。

算法 MatrixChain 的主要计算量取决于程序中对 r, i 和 k 的三重循环。循环体内的计算量为 $O(1)$,而三重循环的总次数为 $O(n^3)$。因此,该算法的计算时间上界为 $O(n^3)$。算法所占用的空间显然为 $O(n^2)$。由此可见,动态规划算法比穷举搜索法要有效得多。

4. 构造最优解

动态规划算法的第4步是构造问题的一个最优解。算法 MatrixChain 只是计算出了最优值,并未给出最优解。也就是说,通过 MatrixChain 的计算,我们只知道要计算所给的矩阵连乘积所需的最少数乘次数,还不知道具体应按什么次序来做矩阵乘法才能达到最少的数乘次数。

然而,MatrixChain 已记录了构造一个最优解所需要的全部信息。事实上,s[i][j] 中的数 k 告诉我们计算矩阵链 $A[i:j]$ 的最佳方式应在矩阵 A_k 和 A_{k+1} 之间断开,即最优的加括号方式应为 $(A[i:k])(A[k+1:j])$。因此,从 s[1][n] 记录的信息可知计算 $A[1:n]$ 的最优加括号方式为 $(A[1:s[1][n]])(A[s[1][n]+1:n])$。而 $A[1:s[1][n]]$ 的最优加括号方式为 $(A[1:s[1][s[1][n]]])(A[s[1][s[1][n]]+1:s[1][s[1][n]]])$。同理可以确定 $A[s[1][n]+1:n]$ 的最优加括号方式在 $s[s[1][n]+1][n]$ 处断开……照此递推下去,最终可以确定 $A[1:n]$ 的最优完全加括号方式,即构造出问题的一个最优解。

下面的算法 Traceback 按算法 MatrixChain 计算出的断点矩阵 s 指示的加括号方式输出计算 $A[i:j]$ 的最优计算次序。

```
void Traceback(int i, int j, int * * s)
{
    if (i == j) return;
    Traceback(i,s[i][j],s);
    Traceback(s[i][j] + 1, j,s);
    cout << "Multiply A" << i << "," << s[i][j];
    cout << " and A" << (s[i][j] + 1) << "," << j
         << endl;
}
```

要输出 $A[1:n]$ 的最优计算次序只要调用 Traceback$(1,n,s)$ 即可。对于上面所举的例子,通过调用 Traceback$(1,6,s)$,即可输出最优计算次序 $((A_1(A_2A_3))((A_4A_5)A_6))$。

3.2 动态规划算法的基本要素

从计算矩阵连乘积最优计算次序的动态规划算法可以看出,该算法的有效性依赖于问题本身所具有的两个重要性质:最优子结构性质和子问题重叠性质。从一般意义上讲,问题所具有的这两个重要性质是该问题可用动态规划算法求解的基本要素。这对于我们在设计求解具体问题的算法时,是否选择动态规划算法具有指导意义。下面我们着重讨论动态规划算法的这两个基本要素以及动态规划法的一个变形——备忘录方法。

1. 最优子结构

设计动态规划算法的第1步通常是要刻画最优解的结构。当问题的最优解包含了其子问题的最优解时,称该问题具有最优子结构性质。问题的最优子结构性质提供了该问题可用动态规划算法求解的重要线索。

在矩阵连乘积最优计算次序问题中,我们注意到,若 $A_1A_2\cdots A_n$ 的最优完全加括号方式在 A_k 和 A_{k+1} 之间将矩阵链断开,则由此确定的子链 $A_1A_2\cdots A_k$ 和 $A_{k+1}A_{k+2}\cdots A_n$ 的完全加括号

方式也最优。即该问题具有最优子结构性质。在分析该问题的最优子结构性质时，我们的方法具有普遍性。首先我们假设由问题的最优解导出的其子问题的解不是最优的，然后再设法说明在这个假设下可构造出一个比原问题最优解更好的解，从而导致矛盾。

在动态规划算法中，问题的最优子结构性质使我们能够以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。同时，它也使我们能在相对小的子问题空间中考虑问题。例如，在矩阵连乘积最优计算次序问题中，子问题空间是输入的矩阵链的所有不同子链组成的。所有不同子链的个数为 $\theta(n^2)$，因而子问题空间的规模为 $\theta(n^2)$。

2. 重叠子问题

可用动态规划算法求解的问题应具备的另一基本要素是子问题的重叠性质。在用递归算法自顶向下解此问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。通常，不同的子问题个数随输入问题的大小呈多项式增长。因此，用动态规划算法通常只需要多项式时间，从而获得较高的解题效率。

为了说明这一点，我们来看在计算矩阵连乘积最优计算次序时，利用递归式直接计算 $\boldsymbol{A}[i:j]$ 的递归算法 RecurMatrixChain。

```
int RecurMatrixChain(int i, int j)
{
  if (i == j) return 0;
  int u = RecurMatrixChain(i,i)
          + RecurMatrixChain(i + 1,j)
          + p[i - 1] * p[i] * p[j];
  s[i][j] = i;
   for (int k = i + 1; k < j; k ++) {
      int t = RecurMatrixChain(i,k)
              + RecurMatrixChain(k + 1,j)
              + p[i - 1] * p[k] * p[j];
      if (t < u) {
        u = t;
       s[i][j] = k;}
      }
   return u;
}
```

用算法 RecurMatrixChain(1,4) 计算 $\boldsymbol{A}[1:4]$ 的递归树如图 3-2 所示。从该图可以看出，许多子问题被重复计算。

事实上，可以证明该算法的计算时间 $T(n)$ 有指数下界。设算法中判断语句和赋值语句花费常数时间，则由算法的递归部分可得关于 $T(n)$ 的递归不等式如下：

$$T(n) \geqslant \begin{cases} O(1) & n = 1 \\ 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1) & n > 1 \end{cases}$$

因此，当 $n > 1$ 时，

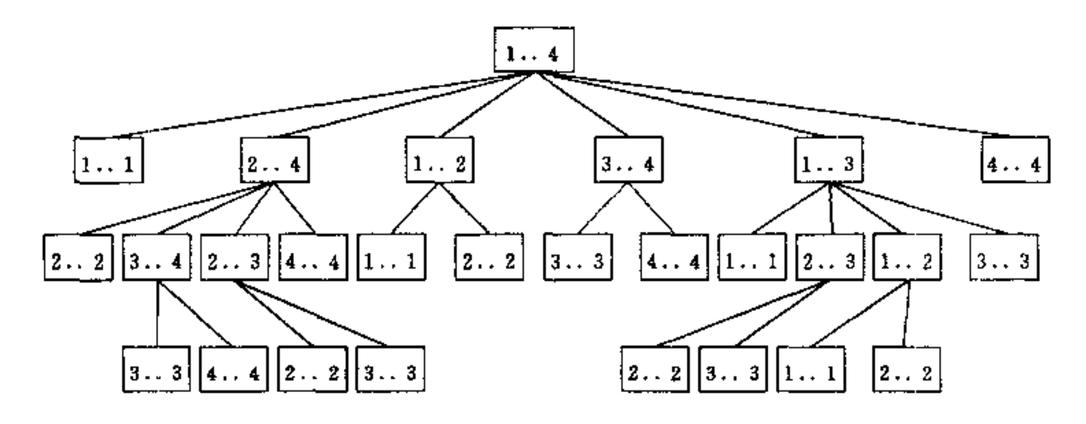


图 3-2 计算 $A[1:4]$ 的递归树

$$T(n) \geqslant 1 + (n-1) + \sum_{k=1}^{n-1} T(k) + \sum_{k=1}^{n-1} T(n-k) = n + 2\sum_{k=1}^{n-1} T(k)$$

据此,可用数学归纳法证明 $T(n) \geqslant 2^{n-1} = \Omega(2^n)$。

因此,直接递归算法 RecurMatrixChain 的计算时间随 n 指数增长。相比之下,解同一问题的动态规划算法 MatrixChain 只需计算时间 $O(n^3)$。其有效性就在于它充分利用了问题的子问题重叠性质。不同的子问题个数为 $\theta(n^2)$,而动态规划算法对于每个不同的子问题只计算一次,从而减少了大量不必要的计算。由此也可看出,在解某一问题的直接递归算法所产生的递归树中,相同的子问题反复出现,并且不同子问题的个数又相对较少时,用动态规划算法是有效的。

3. 备忘录方法

动态规划算法的一个变形是备忘录方法。备忘录方法也用一个表格来保存已解决的子问题的答案,在下次需要解此子问题时,只要简单地查看该子问题的解答,而不必重新计算。与动态规划算法不同的是,备忘录方法的递归方式是自顶向下的,而动态规划算法则是自底向上递归的。因此,备忘录方法的控制结构与直接递归方法的控制结构相同,区别在于备忘录方法为每个解过的子问题建立了备忘录以备需要时查看,避免了相同子问题的重复求解。

备忘录方法为每个子问题建立一个记录项,初始化时,该记录项存入一个特殊的值,表示该子问题尚未求解。在求解过程中,对每个待求的子问题,首先查看其相应的记录项。若记录项中存储的是初始化时存入的特殊值,则表示该子问题是第一次遇到,则此时计算出该子问题的解,并保存在其相应的记录项中。若记录项中存储的已不是初始化时存入的特殊值,则表示该子问题已被计算过,其相应的记录项中存储的是该子问题的解答。此时,只要从记录项中取出该子问题的解答即可。

下面的算法 MemoizedMatrixChain 是解矩阵连乘积最优计算次序问题的备忘录方法。

```
int MemoizedMatrixChain(int n,int * * m, int * * s)
{
  for (int i = 1; i < = n; i++)
    for (int j = i; j < = n; j++)
      m[i][j] = 0;
return LookupChain(1,n);
}
```

```
int LookupChain(int i, int j)
{
  if (m[i][j] > 0) return m[i][j];
  if (i == j) return 0;
  int u = LookupChain(i,i)
       + LookupChain(i + 1,j)
       + p[i - 1] * p[i] * p[j];
  s[i][j] = i;
    for (int k = i + 1; k < j; k ++) {
         int t = LookupChain(i,k)
               + LookupChain(k + 1,j)
               + p[i - 1] * p[k] * p[j];
         if (t < u) {
                   u = t;
                   s[i][j] = k;}
         }
    m[i][j] = u;
    return u;
 }
```

与动态规划算法 MatrixChain 一样,备忘录算法 MemoizedMatrixChain 用数组 m 来记录子问题的最优值。m 初始化为 0,表示相应的子问题还未被计算。在调用 LookupChain 时,若 $m[i][j] > 0$,则表示其中存储的是所要求子问题的计算结果,直接返回此结果即可。否则与直接递归算法一样,自顶向下地递归计算,并将计算结果存入 $m[i][j]$ 后返回。因此,LookupChain 总能返回正确的值,但仅在它第一次被调用时计算,以后的调用就直接返回计算结果。

与动态规划算法一样,备忘录算法 MemoizedMatrixChain 耗时 $O(n^3)$。事实上,共有 $O(n^2)$ 个备忘记录项 $m[i][j], i = 1,\cdots,n; j = i,\cdots,n$。这些记录项的初始化耗费 $O(n^2)$ 时间。每个记录项只填入一次。每次填入时,不包括填入其他记录项的时间,共耗费 $O(n)$ 时间。因此,LookupChain 填入 $O(n^2)$ 个记录项总共耗费 $O(n^3)$ 计算时间。由此可见,通过使用备忘录技术,直接递归算法的计算时间从 $\Omega(2^n)$ 降至 $O(n^3)$。

综上所述,矩阵连乘积的最优计算次序问题可用自顶向下的备忘录算法或自底向上的动态规划算法在 $O(n^3)$ 计算时间内求解。这两个算法都利用了子问题重叠性质。总共有 $\theta(n^2)$ 个不同的子问题。对每个子问题,两种方法都只解一次,并记录答案。再次遇到该子问题时,简单地取用已得到的答案,节省了计算量,提高了算法的效率。

一般来讲,当一个问题的所有子问题都至少要解一次时,则用动态规划算法比用备忘录方法好。此时,动态规划算法没有任何多余的计算,还可利用其规则的表格存取方式,来减少在动态规划算法中的计算时间和空间需求。当子问题空间中的部分子问题可不必求解时,用备忘录方法则较有利,因为从其控制结构可以看出,该方法只解那些确实需要求解的子问题。

3.3 最长公共子序列

一个给定序列的子序列是在该序列中删去若干元素后得到的序列。确切地说,若给定序列

$X = \{x_1, x_2, \cdots, x_m\}$，则另一序列 $Z = \{z_1, z_2, \cdots, z_k\}$，是 X 的子序列是指存在一个严格递增下标序列 $\{i_1, i_2, \cdots, i_k\}$ 使得对于所有 $j = 1, 2, \cdots, k$ 有：$z_j = x_{i_j}$。例如，序列 $Z = \{B, C, D, B\}$ 是序列 $X = \{A, B, C, B, D, A, B\}$ 的子序列，相应的递增下标序列为 $\{2,3,5,7\}$。

给定两个序列 X 和 Y，当另一序列 Z 既是 X 的子序列又是 Y 的子序列时，称 Z 是序列 X 和 Y 的公共子序列。

例如，若 $X = \{A, B, C, B, D, A, B\}$，$Y = \{B, D, C, A, B, A\}$ 则序列 $\{B, C, A\}$ 是 X 和 Y 的一个公共子序列，但它不是 X 和 Y 的一个最长公共子序列。序列 $\{B, C, B, A\}$ 也是 X 和 Y 的一个公共子序列，它的长度为 4，而且它是 X 和 Y 的一个最长公共子序列，因为 X 和 Y 没有长度大于 4 的公共子序列。

最长公共子序列问题：给定两个序列 $X = \{x_1, x_2, \cdots, x_m\}$ 和 $Y = \{y_1, y_2, \cdots, y_n\}$，找出 X 和 Y 的一个最长公共子序列。

动态规划算法可有效地解此问题。下面我们按照动态规划算法设计的步骤来设计一个有效算法。

1. 最长公共子序列的结构

解最长公共子序列问题的最容易想到的算法是穷举搜索法，即对 X 的所有子序列，检查它是否也是 Y 的子序列，从而确定它是否为 X 和 Y 的公共子序列。并且在检查过程中记录最长的公共子序列。X 的所有子序列都检查过后即可求出 X 和 Y 的最长公共子序列。X 的每个子序列相应于下标集 $\{1, 2, \cdots, m\}$ 的一个子集。因此，共有 2^m 个不同子序列，从而穷举搜索法需要指数时间。

事实上，最长公共子序列问题具有最优子结构性质。

设序列 $X = \{x_1, x_2, \cdots, x_m\}$ 和 $Y = \{y_1, y_2, \cdots, y_n\}$ 的一个最长公共子序列为 $Z = \{z_1, z_2, \cdots, z_k\}$，则

(1) 若 $x_m = y_n$，则 $z_k = x_m = y_n$，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的最长公共子序列。

(2) 若 $x_m \neq y_n$ 且 $z_k \neq x_m$，则 Z 是 X_{m-1} 和 Y 的最长公共子序列。

(3) 若 $x_m \neq y_n$ 且 $z_k \neq y_n$，则 Z 是 X 和 Y_{n-1} 的最长公共子序列。

其中，$X_{m-1} = \{x_1, x_2, \cdots, x_{m-1}\}$；$Y_{n-1} = \{y_1, y_2, \cdots, y_{n-1}\}$；$Z_{k-1} = \{z_1, z_2, \cdots, z_{k-1}\}$。

证明：(1) 用反证法。若 $z_k \neq x_m$，则 $\{z_1, z_2, \cdots, z_k, x_m\}$ 是 X 和 Y 的长度为 $k+1$ 的公共子序列。这与 Z 是 X 和 Y 的一个最长公共子序列矛盾。因此，必有 $z_k = x_m = y_n$。由此可知 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个长度为 $k-1$ 的公共子序列。若 X_{m-1} 和 Y_{n-1} 有一个长度大于 $k-1$ 的公共子序列 W，则将 x_m 加在其尾部产生 X 和 Y 的一个长度大于 k 的公共子序列。此为矛盾。故 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个最长公共子序列。

(2) 由于 $z_k \neq x_m$，Z 是 X_{m-1} 和 Y 的一个公共子序列。若 X_{m-1} 和 Y 有一个长度大于 k 的公共子序列 W，则 W 也是 X 和 Y 的一个长度大于 k 的公共子序列。这与 Z 是 X 和 Y 的一个最长公共子序列矛盾。由此即知，Z 是 X_{m-1} 和 Y 的一个最长公共子序列。

(3) 证明与(2)类似。

上述性质告诉我们，两个序列的最长公共子序列包含了这两个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有最优子结构性质。

2. 子问题的递归结构

由最长公共子序列问题的最优子结构性质可知，要找出 $X = \{x_1, x_2, \cdots, x_m\}$ 和

$Y = \{y_1, y_2, \cdots, y_n\}$ 的最长公共子序列,可按以下方式递归地进行:当 $x_m = y_n$ 时,找出 X_{m-1} 和 Y_{n-1} 的最长公共子序列,然后在其尾部加上 $x_m(= y_n)$ 即可得 X 和 Y 的一个最长公共子序列。当 $x_m \neq y_n$ 时,必须解两个子问题,即找出 X_{m-1} 和 Y 的一个最长公共子序列及 X 和 Y_{n-1} 的一个最长公共子序列。这两个公共子序列中较长者即为 X 和 Y 的一个最长公共子序列。

由此递归结构容易看到最长公共子序列问题具有子问题重叠性质。例如,在计算 X 和 Y 的最长公共子序列时,可能要计算 X 和 Y_{n-1} 及 X_{m-1} 和 Y 的最长公共子序列。而这两个子问题都包含一个公共子问题,即计算 X_{m-1} 和 Y_{n-1} 的最长公共子序列。

与矩阵连乘积最优计算次序问题类似,我们来建立子问题最优值的递归关系。用 $c[i][j]$ 记录序列 X_i 和 Y_j 的最长公共子序列的长度。其中,$X_i = \{x_1, x_2, \cdots, x_i\}$;$Y_j = \{y_1, y_2, \cdots, y_j\}$。当 $i = 0$ 或 $j = 0$ 时,空序列是 X_i 和 Y_j 的最长公共子序列。故此时 $c[i][j] = 0$。其他情况下,由最优子结构性质可建立递归关系如下:

$$c[i][j] = \begin{cases} 0 & i = 0, j = 0 \\ c[i-1][j-1] + 1 & i, j > 0; x_i = y_j \\ \max\{c[i][j-1], c[i-1][j]\} & i, j > 0; x_i \neq y_j \end{cases}$$

3. 计算最优值

直接利用递归式容易写出一个计算 $c[i][j]$ 的递归算法,但其计算时间是随输入长度指数增长的。由于在所考虑的子问题空间中,总共有 $\theta(mn)$ 个不同的子问题,因此,用动态规划算法自底向上计算最优值能提高算法的效率。

计算最长公共子序列长度的动态规划算法 LCSLength 以序列 $X = \{x_1, x_2, \cdots, x_m\}$ 和 $Y = \{y_1, y_2, \cdots, y_n\}$ 作为输入,输出两个数组 c 和 b。其中 c[i][j] 存储 X_i 和 Y_j 的最长公共子序列的长度,b[i][j] 记录 c[i][j] 的值是由哪一个子问题的解得到的,这在构造最长公共子序列时要用到。问题的最优值,即 X 和 Y 的最长公共子序列的长度记录于 c[m][n] 中。

```
void LCSLength(int m,int n,char *x,char *y, int **c, Type **b)
{ int i,j;
    for (i = 1; i <= m; i++) c[i][0] = 0;
    for (i = 1; i <= n; i++) c[0][i] = 0;
        for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++) {
            if (x[i] == y[j]) {
                c[i][j] = c[i-1][j-1] + 1;
                b[i][j] = '↖';
                }
            else if (c[i-1][j] >= c[i][j-1]) {
                c[i][j] = c[i-1][j];
                b[i][j] = '↑';
                }
            else    {
                c[i][j] = c[i][j-1];
                b[i][j] = '←';
                }
```

由于每个数组单元的计算耗费 $O(1)$ 时间，算法 LCSLength 耗时 $O(mn)$。

4. 构造最长公共子序列

由算法 LCSLength 计算得到的数组 b 可用于快速构造序列 $X = \{x_1, x_2, \cdots, x_m\}$ 和 $Y = \{y_1, y_2, \cdots, y_n\}$ 的最长公共子序列。首先从 b[m][n] 开始，沿着其中的箭头所指的方向在数组 b 中搜索。当在 b[i][j] 中遇到'↖'时，表示 X_i 和 Y_j 的最长公共子序列是由 X_{i-1} 和 Y_{j-1} 的最长公共子序列在尾部加上 x_i 所得到的子序列。当在 b[i][j] 中遇到'↑'时，表示 X_i 和 Y_j 的最长公共子序列与 X_{i-1} 和 Y_j 的最长公共子序列相同。当在 b[i][j] 中遇到'←'时，表示 X_i 和 Y_j 的最长公共子序列与 X_i 和 Y_{j-1} 的最长公共子序列相同。

下面的算法 LCS 实现根据 b 的内容打印出 X_i 和 Y_j 的最长公共子序列。通过算法调用 LCS(m, n, x, b) 便可打印出序列 X 和 Y 的最长公共子序列。

```
void LCS(int i, int j, char * x, Type * * b)
{
    if (i == 0 || j == 0) return;
    if (b[i][j] == '↖'){
        LCS(i - 1, j - 1, x, b);
        cout << x[i];
        }
    else if (b[i][j] == '↑') LCS(i - 1, j, x, b);
    else LCS(i, j - 1, x, b);
}
```

在算法 LCS 中，每一次递归调用使 i 或 j 减 1，因此算法的计算时间为 $O(m + n)$。

例如，设所给的 2 个序列为 $X = \{A, B, C, B, D, A, B\}$ 和 $Y = \{B, D, C, A, B, A\}$。由算法 LCSLength 和 LCS 计算出的结果如图 3-3 所示。

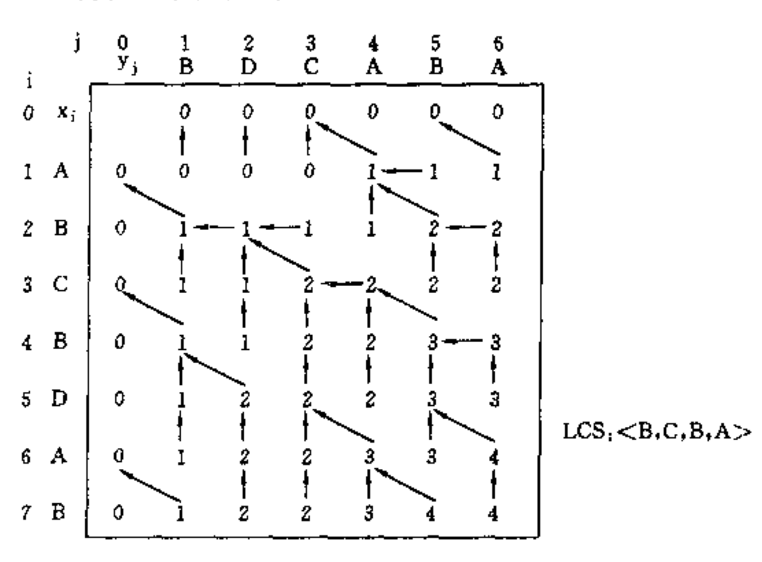


图 3-3　算法 LCS 的计算结果

5. 算法的改进

对于一个具体问题,按照一般的算法设计策略设计出的算法,往往在算法的时间和空间需求上还有较大的改进余地。通常可以利用具体问题的一些特殊性对算法作进一步改进。例如,在算法 LCSLength 和 LCS 中,可进一步将数组 b 省去。事实上,数组元素 $c[i][j]$ 的值仅由 $c[i-1][j-1]$,$c[i-1][j]$ 和 $c[i][j-1]$ 这三个数组元素的值所确定。对于给定的数组元素 $c[i][j]$,我们可以不借助于数组 b 而仅借助于 c 本身在 $O(1)$ 时间内确定 $c[i][j]$ 的值是由 $c[i-1][j-1]$,$c[i-1][j]$ 和 $c[i][j-1]$ 中哪一个值所确定的。因此,我们可以写一个类似于 LCS 的算法,不用数组 b 而在 $O(m+n)$ 时间内构造最长公共子序列。从而可节省 $\theta(mn)$ 的空间。由于数组 c 仍需要 $\theta(mn)$ 的空间,因此,在渐近的意义上,算法仍需要 $\theta(mn)$ 的空间,所作的改进,只是对空间复杂性的常数因子的改进。

另外,如果只需要计算最长公共子序列的长度,则算法的空间需求可大大减少。事实上,在计算 $c[i][j]$ 时,只用到数组 c 的第 i 行和第 $i-1$ 行。因此,用两行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至 $O(\min\{m,n\})$。

3.4 最大子段和

给定由 n 个整数(可能为负整数)组成的序列 $a_1,a_2,\cdots,a_n$,求该序列形如 $\sum_{k=i}^{j} a_k$ 的子段和的最大值。当所有整数均为负整数时定义其最大子段和为 0。依此定义,所求的最优值为

$$\max\left\{0, \max_{1\leq i\leq j\leq n} \sum_{k=i}^{j} a_k\right\}$$

例如,当 $(a_1,a_2,a_3,a_4,a_5,a_6)=(-2,11,-4,13,-5,-2)$ 时,最大子段和为 $\sum_{k=2}^{4} a_k = 20$。

1. 最大子段和问题的简单算法

对于最大子段和问题,有多种求解算法。我们先来讨论一个简单算法。其中用数组 a[] 存储给定的 n 个整数 $a_1,a_2,\cdots,a_n$。

```
int MaxSum(int n, int *a, int& besti, int& bestj)
{
    intsum = 0;
    for (int i = 1;i <= n;i++)
    for (int j = i;j <= n;j++) {
        int thissum = 0;
        for (int k = i;k <= j;k++) thissum += a[k];
        if (thissum > sum) {
            sum = thissum;
            besti = i;
            bestj = j;
            }
        }
    return sum;
}
```

从这个算法的 3 个 for 循环可以看出它所需的计算时间是 $O(n^3)$。事实上，如果我们注意到 $\sum_{k=i}^{j} a_k = a_j + \sum_{k=i}^{j-1} a_k$，则可将算法中的最后一个 for 循环省去，避免重复计算，从而使算法得以改进。改进后的算法可描述为：

```
int MaxSum(int n, int *a, int& besti, int& bestj)
{
    intsum = 0;
    for (int i = 1;i <= n;i++) {
        int thissum = 0;
        for (int j = i;j <= n;j++) {
            thissum += a[j];
            if (thissum > sum) {
                sum = thissum;
                 besti = i;
                 bestj = j;
                 }
            }
        }
    return sum;
}
```

改进后的算法显然只需要 $O(n^2)$ 的计算时间。上述改进是在算法设计技巧上的一个改进，能充分利用已经得到的结果，避免重复计算，节省了计算时间。

2. 最大子段和问题的分治算法

针对最大子段和这个具体问题本身的结构，我们还可以从算法设计的策略上对上述 $O(n^2)$ 计算时间算法进行更进一步的改进。从问题的解的结构可以看出，它适合于用分治法求解。

如果将所给的序列 a[1 : n] 分为长度相等的两段 a[1 : $n/2$] 和 a[$n/2+1$: n]，分别求出这两段的最大子段和，则 a[1 : n] 的最大子段和有三种情形：

(1) a[1 : n] 的最大子段和与 a[1 : $n/2$] 的最大子段和相同。

(2) a[1 : n] 的最大子段和与 a[$n/2+1$: n] 的最大子段和相同。

(3) a[1 : n] 的最大子段和为 $\sum_{k=i}^{j} a_k$，且 $1 \leqslant i \leqslant n/2, n/2+1 \leqslant j \leqslant n$。

(1) 和(2) 这两种情形可递归求得。对于情形(3)，容易看出，a[$n/2$] 与 a[$n/2+1$] 在最优子序列中。因此，我们可以在 a[1 : $n/2$] 中计算出 $s1 = \max\limits_{1 \leqslant i \leqslant n/2} \sum_{k=i}^{n/2} a[k]$，并在 a[$n/2+1$: n] 中计算出 $s2 = \max\limits_{n/2+1 \leqslant i \leqslant n} \sum_{k=n/2+1}^{i} a[k]$。则 $s1 + s2$ 即为出现情形(3) 时的最优值。据此可设计出求最大子段和的分治算法如下：

```
int MaxSubSum(int *a, int left, int right)
{
    intsum = 0;
```

```
        if (left == right)sum = a[left] > 0?a[left]:0;
        else {
            int center = (left + right)/2;
            int leftsum = MaxSubSum(a,left,center);
            int rightsum = MaxSubSum(a,center + 1,right);
            int s1 = 0;
            int lefts = 0;
            for (int i = center;i >= left;i--) {
                lefts += a[i];
                if (lefts > s1)s1 = lefts;
                }
            int s2 = 0;
            int rights = 0;
            for (int i = center + 1;i <= right;i++) {
                  rights += a[i];
                  if (rights > s2)s2 = rights;
                  }
            sum = s1 + s2;
             if (sum < leftsum)sum = leftsum;
             if (sum < rightsum)sum = rightsum;
             }
        return sum;
    }

    int MaxSum(int n, int *a)
    {
      return MaxSubSum(a,1,n);
    }
```

该算法所需的计算时间 $T(n)$ 满足典型的分治算法递归式

$$T(n) = \begin{cases} O(1) & n \leqslant c \\ 2T(n/2) + O(n) & n > c \end{cases}$$

解此递归方程可知，$T(n) = O(n\log n)$。

3. 最大子段和问题的动态规划算法

在对上述分治算法的分析中我们注意到，若记 $b[j] = \max\limits_{1 \leqslant i \leqslant j}\left\{\sum\limits_{k=i}^{j} a[k]\right\}, 1 \leqslant j \leqslant n$，则所求的最大子段和为

$$\max_{1 \leqslant i \leqslant j \leqslant n} \sum_{k=i}^{j} a[k] = \max_{1 \leqslant j \leqslant n} \max_{1 \leqslant i \leqslant j} \sum_{k=i}^{j} a[k] = \max_{1 \leqslant j \leqslant n} b[j]$$

由 $b[j]$ 的定义易知，当 $b[j-1] > 0$ 时 $b[j] = b[j-1] + a[j]$，否则 $b[j] = a[j]$。由此可得计算 $b[j]$ 的动态规划递归式

$$b[j] = \max\{b[j-1] + a[j], a[j]\}, \quad 1 \leqslant j \leqslant n$$

据此，可设计出求最大子段和的动态规划算法如下：

```
int MaxSum(int n, int * a)
{
    intsum = 0,
        b = 0;
    for (int i = 1;i <= n;i++) {
        if (b > 0) b += a[i];
        else b = a[i];
        if (b > sum)sum = b;
        }
    return sum;
}
```

上述算法显然需要 $O(n)$ 计算时间和 $O(n)$ 空间。

4. 最大子段和问题与动态规划算法的推广

最大子段和问题可以很自然地推广到高维的情形。

(1) 最大子矩阵和问题:给定一个 m 行 n 列的整数矩阵 $\boldsymbol{A}$,试求矩阵 $\boldsymbol{A}$ 的一个子矩阵,使其各元素之和为最大。

最大子矩阵和问题是最大子段和问题向二维的推广。用二维数组 a[1:m][1:n] 表示给定的 m 行 n 列的整数矩阵。子数组 a[$i1$:$i2$][$j1$:$j2$] 表示左上角和右下角行列坐标分别为($i1$,$j1$)和($i2$,$j2$)的子矩阵,其各元素之和记为

$$s(i1,i2,j1,j2) = \sum_{i=i1}^{i2}\sum_{j=j1}^{j2} a[i][j]$$

最大子矩阵和问题的最优值为 $\max\limits_{\substack{1\le i1\le i2\le m\\1\le j1\le j2\le n}} s(i1,i2,j1,j2)$。

如果用直接枚举的方法解最大子矩阵和问题,需要 $O(m^2n^2)$ 时间。注意到

$$\max_{\substack{1\le i1\le i2\le m\\1\le j1\le j2\le n}} s(i1,i2,j1,j2) = \max_{1\le i1\le i2\le m}\left\{\max_{1\le j1\le j2\le n} s(i1,i2,j1,j2)\right\} = \max_{1\le i1\le i2\le m} t(i1,i2)$$

其中,$t(i1,i2) = \max\limits_{1\le j1\le j2\le n} s(i1,i2,j1,j2) = \max\limits_{1\le j1\le j2\le n}\sum\limits_{j=j1}^{j2}\sum\limits_{i=i1}^{i2} a[i][j]$。

设 $b[j] = \sum\limits_{i=i1}^{i2} a[i][j]$,则 $t(i1,i2) = \max\limits_{1\le j1\le j2\le n}\sum\limits_{j=j1}^{j2} b[j]$。

容易看出,这正是一维情形的最大子段和问题。由此,借助于最大子段和问题的动态规划算法 MaxSum,可设计出解最大子矩阵和问题的动态规划算法 MaxSum2 如下:

```
int MaxSum2(int m, int n, int ** a)
{
    intsum = 0;
    int *b = new int [n + 1];
    for (int i = 1;i <= m;i++) {
        for (int k = 1;k <= n;k++) b[k] = 0;
        for (int j = i;j <= m;j++) {
            for (int k = 1;k <= n;k++) b[k] += a[j][k];
            int max = MaxSum(n,b);
            if (max > sum)sum = max;
```

```
            }
        }
    return sum;
}
```

由于解最大子段和问题的动态规划算法 MaxSum 需要 $O(n)$ 时间,故算法 MaxSum2 的双重 for 循环需要 $O(m^2 n)$ 计算时间。从而算法 MaxSum2 需要 $O(m^2 n)$ 计算时间。特别地,当 $m = O(n)$ 时,算法 MaxSum2 需要 $O(n^3)$ 计算时间。

(2) 最大 m 子段和问题:给定由 n 个整数(可能为负整数)组成的序列 $a_1, a_2, \cdots, a_n$,以及一个正整数 m,要求确定序列 $a_1, a_2, \cdots, a_n$ 的 m 个不相交子段,使这 m 个子段的总和达到最大。

最大 m 子段和问题是最大子段和问题在子段个数上的推广。换句话说,最大子段和问题是最大 m 子段和问题当 $m = 1$ 时的特殊情形。

设 $b(i,j)$ 表示数组 a 的前 j 项中 i 个子段和的最大值,且第 i 个子段含 $a[j]$ $(1 \leqslant i \leqslant m, i \leqslant j \leqslant n)$。则所求的最优值显然为 $\max\limits_{m \leqslant j \leqslant n} b(m,j)$。与最大子段和问题类似地,计算 $b(i,j)$ 的递归式为

$$b(i,j) = \max\{b(i,j-1) + a[j], \max_{i-1 \leqslant t < j} b(i-1,t) + a[j]\} \quad (1 \leqslant i \leqslant m, i \leqslant j \leqslant n)$$

其中,$b(i,j-1) + a[j]$ 项表示第 i 个子段含 $a[j-1]$,而 $\max\limits_{i-1 \leqslant t < j} b(i-1,t) + a[j]$ 项表示第 i 个子段仅含 $a[j]$。初始时,$b(0,j) = 0, (1 \leqslant j \leqslant n)$;$b(i,0) = 0, (1 \leqslant i \leqslant m)$。

根据上述计算 $b(i,j)$ 的动态规划递归式,可设计解最大 m 子段和问题的动态规划算法如下:

```
int MaxSum(int m, int n, int *a)
{
    if (n < m || m < 1) return 0;
    int **b = new int *[m + 1];
    for (int i = 0; i <= m; i++)
      b[i] = new int [n + 1];
    for (int i = 0; i <= m; i++) b[i][0] = 0;
    for (int j = 1; j <= n; j++) b[0][j] = 0;
    for (int i = 1;i <= m;i++)
      for (int j = i;j <= n - m + i;j++)
        if (j > i) {
          b[i][j] = b[i][j - 1] + a[j];
          for (int k = i - 1;k < j;k++)
            if (b[i][j] < b[i - 1][k] + a[j])
              b[i][j] = b[i - 1][k] + a[j];
          }
        else b[i][j] = b[i - 1][j - 1] + a[j];
      int sum = 0;
      for (int j = m;j <= n;j++)
        if (sum < b[m][j]) sum = b[m][j];
      return sum;
}
```

上述算法显然需要 $O(mn^2)$ 计算时间和 $O(mn)$ 空间。

注意到在上述算法中，计算 b[i][j] 时只用到数组 b 的第 $i-1$ 行和第 i 行的值。因而算法中只要存储数组 b 的当前行，不必存储整个数组。另一方面，$\max\limits_{i-1\leq t<j} b(i-1,t)$ 的值可以在计算第 $i-1$ 行时预先计算并保存起来。这样一来，在计算第 i 行的值时不必重新计算，节省了计算时间和空间。按此思想可对上述算法作进一步改进如下：

```
int MaxSum(int m, int n, int *a)
{
    if (n < m || m < 1) return 0;
    int *b = new int [n + 1];
    int *c = new int [n + 1];
    b[0] = 0;
    c[1] = 0;
    for (int i = 1;i <= m;i++) {
      b[i] = b[i - 1] + a[i];
      c[i - 1] = b[i];
      int max = b[i];
      for (int j = i + 1;j <= i + n - m;j++) {
            b[j] = b[j - 1] > c[j - 1]?b[j - 1] + a[j]:c[j - 1] + a[j];
            c[j - 1] = max;
            if (max < b[j]) max = b[j];
            }
      c[i + n - m] = max;
      }
    intsum = 0;
    for (int j = m;j <= n;j++)
      if (sum < b[j])sum = b[j];
    return sum;
}
```

上述算法需要 $O(m(n-m))$ 计算时间和 $O(n)$ 空间。当 m 或 $n-m$ 为常数时，上述算法需要 $O(n)$ 计算时间和 $O(n)$ 空间。

3.5 凸多边形最优三角剖分

用动态规划算法能有效地解凸多边形的最优三角剖分问题。尽管这是一个几何问题，但在本质上它与矩阵连乘积的最优计算次序问题极为相似。

多边形是平面上一条分段线性闭曲线。也就是说，多边形是由一系列首尾相接的直线段所组成的。组成多边形的各直线段称为该多边形的边。连接多边形相继两条边的点称为多边形的顶点。若多边形的边除了连接顶点外没有别的交点，则称该多边形为一简单多边形。一个简单多边形将平面分为三个部分：被包围在多边形内的所有点构成了多边形的内部；多边形本身构成多边形的边界；而平面上其余包围着多边形的点构成了多边形的外部。当一个简单多边形及其内部构成一个闭凸集时，称该简单多边形为一凸多边形。即凸多边形边界上或内部的任意两点所连成的直线段上所有点均在凸多边形的内部或边界上。

通常，用多边形顶点的逆时针序列来表示一个凸多边形，即 $P=\{v_0,v_1,\cdots,v_{n-1}\}$ 表示具

有 n 条边 $v_0v_1, v_1v_2, \cdots, v_{n-1}v_n$,的一个凸多边形。其中,约定 $v_0 = v_n$。

若 v_i 与 v_j 是多边形上不相邻的两个顶点,则线段 v_iv_j 称为多边形的一条弦。弦 v_iv_j 将多边形分割成两个多边形 $\{v_i, v_{i+1}, \cdots, v_j\}$ 和 $\{v_j, v_{j+1}, \cdots, v_i\}$。

多边形的三角剖分是一个将多边形分割成互不相交的三角形的弦的集合 T。图 3-4 是一个凸 7 边形的两个不同的三角剖分。

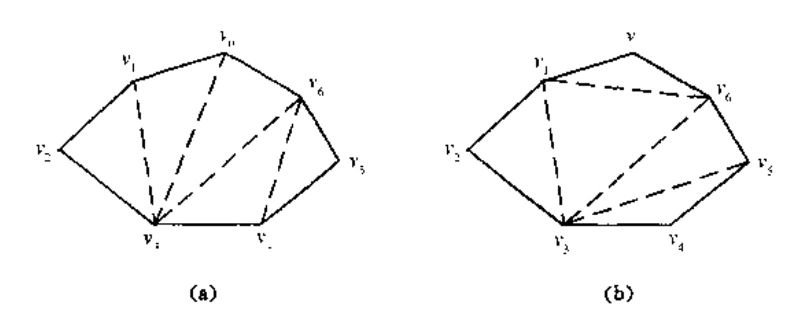


图 3-4　一个凸 7 边形的两个不同的三角剖分

在凸多边形 P 的一个三角剖分 T 中,各弦互不相交,且集合 T 已达到最大,即 P 的任一不在 T 中的弦必与 T 中某一弦相交。在一个有 n 个顶点的凸多边形的三角剖分中,恰有 $n-3$ 条弦和 $n-2$ 个三角形。

凸多边形最优三角剖分问题:给定一个凸多边形 $P = \{v_0, v_1, \cdots, v_{n-1}\}$,以及定义在由凸多边形的边和弦组成的三角形上的权函数 w。要求确定该凸多边形的一个三角剖分,使得该三角剖分中诸三角形上权之和为最小。

可以定义三角形上各种各样的权函数 w。例如:$w(v_iv_jv_k) = |v_iv_j| + |v_jv_k| + |v_kv_i|$,其中,$|v_iv_j|$ 是点 v_i 到 v_j 的欧氏距离。相应于此权函数的最优三角剖分即为最小弦长三角剖分。

本节所述算法可适用于任意权函数。

1. 三角剖分的结构及其相关问题

凸多边形的三角剖分与表达式的完全加括号方式之间具有十分紧密的联系。正如所看到的,矩阵连乘积的最优计算次序问题等价于矩阵链的最优完全加括号方式。这些问题之间的相关性可从它们所对应的完全二叉树的同构性看出。

一个表达式的完全加括号方式相应于一棵完全二叉树,称为表达式的语法树。例如,完全加括号的矩阵连乘积 $((A_1(A_2A_3))(A_4(A_5A_6)))$ 所相应的语法树如图 3-5(a) 所示。

语法树中每一个叶结点表示表达式中一个原子。在语法树中,若一结点有一个表示表达式 E_l 的左子树,以及一个表示表达式 E_r 的右子树,则以该结点为根的子树表示表达式 (E_lE_r)。因此,有 n 个原子的完全加括号表达式对应于惟一的一棵有 n 个叶结点的语法树,反之亦然。

凸多边形 $\{v_0, v_1, \cdots, v_{n-1}\}$ 的三角剖分也可以用语法树来表示。例如,图 3-5(a) 中凸多边形的三角剖分可用图 3-5(b) 所示的语法树来表示。该语法树的根结点为边 v_0v_6。三角剖分中的弦组成其余的内结点。多边形中除 v_0v_6 边外的各边都是语法树的一个叶结点。树根 v_0v_6 是三角形 $v_0v_3v_6$ 的一条边。该三角形将原凸多边形分为三个部分:三角形 $v_0v_3v_6$,凸多边形 $\{v_0, v_1, \cdots, v_3\}$ 和凸多边形 $\{v_3, v_4, \cdots, v_6\}$,三角形 v_0, v_3, v_6 的另外两条边,即弦 v_0v_3 和 v_3v_6 为根的两个儿子。以它们为根的子树表示凸多边形 $\{v_0, v_1, \cdots, v_3\}$ 和 $\{v_3, v_4, \cdots, v_6\}$ 的三角剖分。

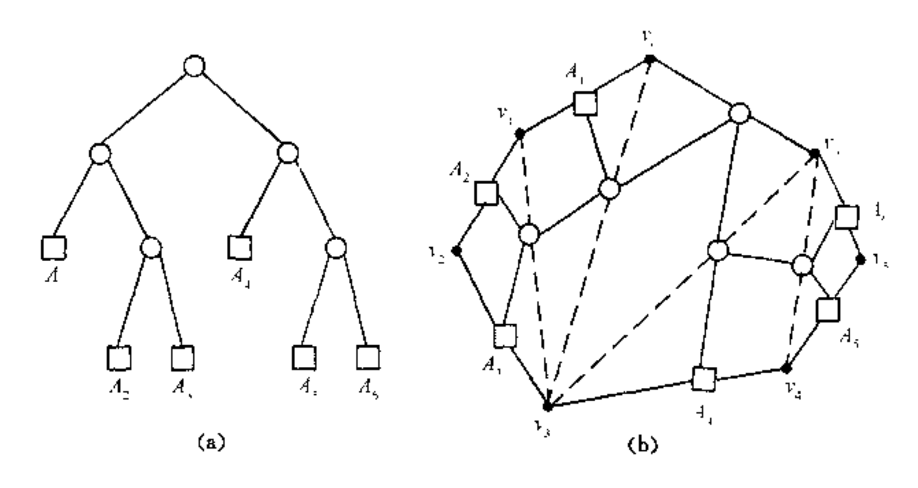


图 3-5　表达式语法树与三角剖分的对应

在一般情况下，一个凸 n 边形的三角剖分对应于一棵有 $n-1$ 个叶结点的语法树。反之，也可根据一棵有 $n-1$ 个叶结点的语法树产生一个相应的凸 n 边形的三角剖分。也就是说，凸 n 边形的三角剖分与有 $n-1$ 个叶结点的语法树之间存在一一对应关系。由于 n 个矩阵的完全加括号乘积与 n 个叶结点的语法树之间存在一一对应关系，因此，n 个矩阵的完全加括号乘积也与凸$(n+1)$边形中的三角剖分之间存在一一对应关系。图 3-5 的(a)和(b)表示出了这种对应关系。矩阵连乘积 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_n$ 中的每个矩阵$\boldsymbol{A}_i$ 对应于凸$(n+1)$边形中的一条边 $v_{i-1}v_i$。三角剖分中的一条弦 v_iv_j，$i<j$，对应于矩阵连乘积 $\boldsymbol{A}[i+1:j]$。

事实上，矩阵连乘积的最优计算次序问题是凸多边形最优三角剖分问题的一个特殊情形。对于给定的矩阵链 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_n$，定义一个与之相应的凸$(n+1)$边形 $P=\{v_0,v_1,\cdots,v_n\}$，使得矩阵 $\boldsymbol{A}_i$ 与凸多边形的边 $v_{i-1}v_i$ 一一对应。若矩阵 $\boldsymbol{A}_i$ 的维数为 $p_{i-1}\times p_i$，$i=1,2,\cdots n$，则定义三角形 $v_iv_jv_k$ 上的权函数值为：$w(v_iv_jv_k)=p_ip_jp_k$。依此权函数的定义，凸多边形 P 的最优三角剖分所对应的语法树给出矩阵链 $\boldsymbol{A}_1\boldsymbol{A}_2\cdots\boldsymbol{A}_n$ 的最优完全加括号方式。

2. 最优子结构性质

凸多边形的最优三角剖分问题有最优子结构性质。

事实上，若凸$(n+1)$边形 $P=\{v_0,v_1,\cdots,v_n\}$ 的一个最优三角剖分 T 包含三角形 $v_0v_kv_n$，$1\leqslant k\leqslant n-1$，则 T 的权为三个部分权的和：三角形 $v_0v_kv_n$ 的权，子多边形$\{v_0,v_1,\cdots,v_k\}$和$\{v_k,k_{k+1},\cdots,v_n\}$的权之和。可以断言，由 T 所确定的这两个子多边形的三角剖分也是最优的。因为若有$\{v_0,v_1,\cdots,v_k\}$或$\{v_k,v_{k+1},\cdots,v_n\}$的更小权的三角剖分将导致 T 不是最优三角剖分的矛盾。

3. 最优三角剖分的递归结构

首先，定义 $t[i][j]$，$1\leqslant i<j\leqslant n$ 为凸子多边形$\{v_{i-1},v_i,\cdots,v_j\}$的最优三角剖分所对应的权函数值，即其最优值。为方便起见，设退化的多边形$\{v_{i-1},v_i\}$具有权值 0。据此定义，要计算的凸$(n+1)$边形 P 的最优权值为 $t[1][n]$。

$t[i][j]$ 的值可以利用最优子结构性质递归地计算。由于退化的两顶点多边形的权值为 0，所以 $t[i][i]=0$，$i=1,2,\cdots,n$。当 $j-i\geqslant 1$ 时，凸子多边形$\{v_{i-1},v_i,\cdots,v_j\}$至少有 3 个顶点。由最优子结构性质，$t[i][j]$ 的值应为 $t[i][k]$ 的值加上 $t[k+1][j]$ 的值，再加上三角

形 $v_{i-1}v_kv_j$ 的权值，其中，$i \leqslant k \leqslant j-1$。由于在计算时还不知道 k 的确切位置，而 k 的所有可能位置只有 $j-i$ 个，因此可以在这 $j-i$ 个位置中选出使 $t[i][j]$ 值达到最小的位置。由此，$t[i][j]$ 可递归地定义为

$$t[i][j] = \begin{cases} 0 & i = j \\ \min\limits_{i \leqslant k < j} \{ t[i][k] + t[k+1][j] + w(v_{i-1}v_kv_j) \} & i < j \end{cases}$$

4. 计算最优值

与矩阵连乘积问题中计算 $m[i][j]$ 的递归式进行比较容易看出，除了权函数的定义外，$t[i][j]$ 与 $m[i][j]$ 的递归式是完全一样的。因此只要对计算 $m[i][j]$ 的算法 MatrixChain 作很小的修改就完全适用于计算 $t[i][j]$。

下面描述的计算凸$(n+1)$边形 $P=\{v_0,v_1,\cdots,v_n\}$ 的最优三角剖分的动态规划算法 MinWeightTriangulation 以凸多边形 $P=\{v_0,v_1,\cdots,v_n\}$ 和定义在三角形上的权函数 w 作为输入。

```
template < class Type >
void MinWeightTriangulation (int n, Type * * t, int * * s)
{
    for (int i = 1; i < = n; i ++) t[i][i] = 0;
    for (int r = 2; r < = n; r ++)
        for (int i = 1; i < = n - r + 1; i ++) {
            int j = i + r - 1;
            t[i][j] = t[i + 1][j] + w(i - 1,i,j);
            s[i][j] = i;
            for (int k = i + 1; k < i + r - 1; k ++) {
                int u = t[i][k] + t[k + 1][j] + w(i - 1,k,j);
                if (u < t[i][j]) {
                    t[i][j] = u;
                    s[i][j] = k;}
                }
            }
}
```

与算法 MatrixChain 一样，算法 MinWeightTriangulation 占用 $O(n^2)$ 空间，耗时 $O(n^3)$。

5. 构造最优三角剖分

算法 MinWeightTriangulation 在计算每一个凸子多边形 $\{v_{i-1},v_i,\cdots,v_j\}$ 的最优值时，用数组 s 记录了最优三角剖分中所有三角形信息。$s[i][j]$ 记录了与 v_{i-1} 和 v_j 一起构成三角形的第 3 个顶点的位置。据此，用 $O(n)$ 时间就可构造出最优三角剖分中的所有三角形。

3.6 多边形游戏

多边形游戏问题是 1998 年国际信息学奥林匹克竞赛试题。

多边形游戏是一个单人玩的游戏，开始时有一个由 n 个顶点构成的多边形。每个顶点被赋予一个整数值，每条边被赋予一个运算符“+”或“*”。所有边依次用整数从 1 到 n 编号。

游戏第1步,将一条边删除。

随后 $n-1$ 步按以下方式操作:

(1) 选择一条边 E 以及由 E 连接着的2个顶点 $v1$ 和 $v2$;

(2) 用一个新的顶点取代边 E 以及由 E 连接着的2个顶点 $v1$ 和 $v2$。将由顶点 $v1$ 和 $v2$ 的整数值通过边 E 上的运算得到的结果赋予新顶点。

最后,所有边都被删除,游戏结束。游戏的得分就是所剩顶点上的整数值。

编程任务:对于给定的多边形,编程计算出最高得分,并且列出所有得到这个最高得分首次被删除的边的编号。

数据输入:由文件 POLYGON.IN 提供输入数据,文件的第一行是所给多边形的顶点数 n;第2行包含所有边1到 n 所对应的运算符,以及与相邻两边相关联的顶点的数值(1号边与2号边之间是1号顶点的数值,2号边与3号边之间是2号顶点的数值,…,依此类推。最后的一个数值对应于与 n 号边和1号边相关联的顶点)。运算符与数值之间由一个空格分隔。字母 t 代表运算符"+",字母 x 代表运算符"*"。文件名由键盘输入。

结果输出:程序运行结束时,将计算结果写入文件 POLYGON.OUT 中。文件的第1行是计算出的最高得分。第2行是所有得到这个最高得分首次被删除的边按升序排列的编号。

输入文件示例	输出文件示例
POLYGON.IN	POLYGON.OUT
4	33
t -7 t 4 x 2 x 5	1 2

该问题与上一节中讨论过的凸多边形最优三角剖分问题类似,但二者的最优子结构性质不同。多边形游戏问题的最优子结构性质更具有一般性。

1. 最优子结构性质

设所给的多边形的顶点和边的顺时针序列为

$$\mathrm{op}[1], v[1], \mathrm{op}[2], v[2], \cdots, \mathrm{op}[n], v[n]$$

其中, $\mathrm{op}[i]$ 表示第 i 条边所相应的运算符, $v[i]$ 表示第 i 个顶点上的数值, $i=1\sim n$。

在所给多边形中,从顶点 $i(1\leqslant i\leqslant n)$ 开始,长度为 j(链中有 j 个顶点)的顺时针链 $p(i,j)$ 可表示为

$$v[i], \mathrm{op}[i+1], \cdots, v[i+j-1]$$

如果这条链的最后一次合并运算在 $\mathrm{op}[i+s]$ 处发生($1\leqslant s\leqslant j-1$),则可在 $\mathrm{op}[i+s]$ 处将链分割为两个子链 $p(i,s)$ 和 $p(i+s,j-s)$。

设 $m1$ 是对子链 $p(i,s)$ 的任意一种合并方式得到的值,而 a 和 b 分别是在所有可能的合并中得到的最小值和最大值。$m2$ 是 $p(i+s,j-s)$ 的任意一种合并方式得到的值,而 c 和 d 分别是在所有可能的合并中得到的最小值和最大值。依此定义我们有

$$a\leqslant m1\leqslant b, c\leqslant m2\leqslant d$$

由于子链 $p(i,s)$ 和 $p(i+s,j-s)$ 的合并方式决定了 $p(i,j)$ 在 $\mathrm{op}[i+s]$ 处断开后的合并方式,在 $\mathrm{op}[i+s]$ 处合并后其值为

$$m=(m1)\mathrm{op}[i+s](m2)$$

(1) 当 $\mathrm{op}[i+s]='+'$ 时,显然有

$$a + c \leqslant m \leqslant b + d$$

换句话说，由链 $p(i,j)$ 合并的最优性可推出子链 $p(i,s)$ 和 $p(i+s,j-s)$ 的最优性，且最大值对应于子链的最大值，最小值对应于子链的最小值。

(2) 当 op[$i+s$] = '*' 时，情况有所不同。由于 $v[i]$ 可取负整数，子链的最大值相乘未必能得到主链的最大值。但是我们注意到最大值一定在边界点达到，即

$$\min\{ac, ad, bc, bd\} \leqslant m \leqslant \max\{ac, ad, bc, bd\}$$

换句话说，主链的最大值和最小值可由子链的最大值和最小值得到。例如，当 $m = ac$ 时，最大主链由它的两条最小子链组成；同理当 $m = bd$ 时，最大主链由它的两条最大子链组成。无论哪种情形发生，由主链的最优性均可推出子链的最优性。

综上可知多边形游戏问题满足最优子结构性质。

2. 递归求解

由前面的分析可知，为了求链合并的最大值，必须同时求子链合并的最大值和最小值。因此在整个计算过程中，应同时计算最大值和最小值。

设 $m[i,j,0]$ 是链 $p(i,j)$ 合并的最小值，而 $m[i,j,1]$ 是最大值。若最优合并在 $op[i+s]$ 处将 $p(i,j)$ 分为 2 个长度小于 j 的子链 $p(i,i+s)$ 和 $p(i+s,j-s)$，且从顶点 i 开始的长度小于 j 的子链的最大值和最小值均已计算出。为叙述方便，记

$a = m[i,i+s,0]$，$b = m[i,i+s,1]$，$c = m[i+s,j-s,0]$，$d = m[i+s,j-s,1]$。

(1) 当 op[$i+s$] = '+' 时，

$$m[i,j,0] = a + c$$
$$m[i,j,1] = b + d$$

(2) 当 op[$i+s$] = '*' 时，

$$m[i,j,0] = \min\{ac, ad, bc, bd\}$$
$$m[i,j,1] = \max\{ac, ad, bc, bd\}$$

综合(1)和(2)，将 $p(i,j)$ 在 op[$i+s$] 处断开的最大值记为 maxf(i,j,s)，最小值记为 minf(i,j,s)，则

$$\text{minf}(i,j,s) = \begin{cases} a + c & \text{op}[i+s] = '+' \\ \min\{ac, ad, bc, bd\} & \text{op}[i+s] = '*' \end{cases}$$

$$\text{maxf}(i,j,s) = \begin{cases} b + d & \text{op}[i+s] = '+' \\ \max\{ac, ad, bc, bd\} & \text{op}[i+s] = '*' \end{cases}$$

由于最优断开位置 s 有 $1 \leqslant s \leqslant j-1$ 的 $j-1$ 种情况，由此可知

$$m[i,j,0] = \min_{1 \leqslant s \leqslant j}\{\text{minf}(i,j,s)\},\ 1 \leqslant i,j \leqslant n$$

$$m[i,j,1] = \max_{1 \leqslant s \leqslant j}\{\text{maxf}(i,j,s)\},\ 1 \leqslant i,j \leqslant n$$

初始边界值显然为

$$m[i,1,0] = v[i], 1 \leqslant i \leqslant n$$
$$m[i,1,1] = v[i], 1 \leqslant i \leqslant n$$

由于多边形是封闭的，在上面的计算中，当 $i+s > n$ 时，顶点 $i+s$ 实际编号为 $(i+s) \bmod n$。按上述递推式计算出的 $m[i,n,1]$ 即为游戏首次删去第 i 条边后得到的最大得分。

3. 算法描述

基于以上讨论可设计解多边形游戏问题的动态规划算法如下：

```
void MIN_ MAX(int n, int i, ints, int j, int& minf, int& maxf)
{
  int e[4];
  int a = m[i][s][0],
    b = m[i][s][1],
    r = (i + s - 1)%n + 1,
    c = m[r][j - s][0],
    d = m[r][j - s][1];
  if (op[r] = = 't') {
      minf = a + c;
      maxf = b + d;
      }
  else
    {
      e[1] = a * c;
      e[2] = a * d;
      e[3] = b * c;
      e[4] = b * d;
      minf = e[1];
      maxf = e[1];
      for (int r = 2;r < 5;r ++) {
            if (minf > e[r]) minf = e[r];
            if (maxf < e[r]) maxf = e[r];
         }
      }
}

int Poly_ Max(int n)
{
  int minf, maxf;
  for (int j = 2;j < = n;j ++)
    for (int i = 1;i < = n;i ++)
      for (ints = 1;s < j;s ++) {
           MIN_ MAX(n,i,s,j,minf,maxf,m,op);
           if (m[i][j][0] > minf) m[i][j][0] = minf;
           if (m[i][j][1] < maxf) m[i][j][1] = maxf;
         }
  int temp = m[1][n][1];
  for (int i = 2;i < = n;i ++)
    if (temp < m[i][n][1]) temp = m[i][n][1];
  return temp;
}
```

4. 计算复杂性分析

与凸多边形最优三角剖分问题类似，上述算法需要 $O(n^3)$ 计算时间。

3.7 图像压缩

在计算机中常用像素点灰度值序列$\{p_1,p_2,\cdots,p_n\}$表示图像。其中整数 p_i，$1 \leqslant i \leqslant n$，表示像素点 i 的灰度值。通常灰度值的范围是 0 ~ 255。因此，需要用 8 位表示一个像素。

图像的变位压缩存储格式将所给的像素点序列$\{p_1,p_2,\cdots,p_n\}$分割成 m 个连续段 S_1，$S_2,\cdots,S_m$。第 i 个像素段 S_i 中$(1 \leqslant i \leqslant m)$，有 $l[i]$ 个像素，且该段中每个像素都只用 $b[i]$ 位来表示。设 $t[i] = \sum_{k=1}^{i-1} l[k]$，$1 \leqslant i \leqslant m$，则第 i 个像素段 S_i 为

$$S_i = \{p_{t[i]+1},\cdots,p_{t[i]+l[i]}\},1 \leqslant i \leqslant m$$

设 $h_i = \lceil \log(\max\limits_{t[i]+1 \leqslant k \leqslant t[i]+l[i]} p_k + 1)\rceil$，则 $h_i \leqslant b[i] \leqslant 8$。因此需要用 3 位来表示 $b[i]$，$1 \leqslant i \leqslant m$。如果限制 $1 \leqslant l[i] \leqslant 255$，则需要用 8 位来表示 $l[i]$，$1 \leqslant i \leqslant m$。这样一来，第 i 个像素段所需的存储空间为 $l[i] * b[i] + 11$ 位。因此，按此格式存储像素序列$\{p_1,p_2,\cdots,p_n\}$，需要 $\sum_{i=1}^{m} l[i] * b[i] + 11m$ 位的存储空间。

图像压缩问题要求确定像素序列$\{p_1,p_2,\cdots,p_n\}$的一个最优分段，使得依此分段所需的存储空间最少。其中，$0 \leqslant p_i \leqslant 256$，$1 \leqslant i \leqslant n$，每个分段的长度不超过 256 位。

1. 最优子结构性质

设 $l[i]$，$b[i]$，$1 \leqslant i \leqslant m$ 是$\{p_1,p_2,\cdots,p_n\}$的一个最优分段。显而易见，$l[1]$，$b[1]$ 是$\{p_1,\cdots,p_{l[1]}\}$的一个最优分段，且 $l[i]$，$b[i]$，$2 \leqslant i \leqslant m$ 是$\{p_{l[1]+1},\cdots,p_n\}$的一个最优分段。即图像压缩问题满足最优子结构性质。

2. 递归计算最优值

设 $s[i]$，$1 \leqslant i \leqslant n$ 是像素序列$\{p_1,\cdots,p_i\}$的最优分段所需的存储位数。由最优子结构性质易知：

$$s[i] = \min_{1 \leqslant k \leqslant \min\{i,256\}} \{s[i-k] + k * b\max(i-k+1,i)\} + 11$$

其中，$b\max(i,j) = \lceil \log(\max\limits_{i \leqslant k \leqslant j}\{p_k\} + 1)\rceil$。

据此可设计解图像压缩问题的动态规划算法如下：

```
void Compress(int n, int p[],int s[], int l[],int b[])
{
    int Lmax = 256, header = 11;
    s[0] = 0;
    for (int i = 1; i <= n; i++) {
        b[i] = length(p[i]);
        int bmax = b[i];
        s[i] = s[i - 1] + bmax;
```

```
            l[i] = 1;
            for (int j = 2; j <= i && j <= Lmax; j++) {
                if (bmax < b[i - j + 1]) bmax = b[i - j + 1];
                if (s[i] > s[i - j] + j * bmax) {
                        s[i] = s[i - j] + j * bmax;
                        l[i] = j;
                        }
                }
            s[i] += header;
            }
    }

    int length(int i)
    {
      int k = 1;
      i = i/2;
      while (i > 0) {
        k++;
        i = i/2;
        }
        return k;
    }
```

3. 构造最优解

算法 Compress 中用 l[i],b[i] 记录了最优分段所需的信息。最优分段的最后一段的段长度和像素位数分别存储于 l[n] 和 b[n] 中。其前一段的段长度和像素位数存储于 l[n - l[n]] 和 b[n - l[n]] 中。依次类推,由算法计算出的 l 和 b 可在 $O(n)$ 时间内构造出相应的最优解。具体算法可实现如下:

```
    void Traceback(int n, int& i, int s[], int l[])
    {
        if (n == 0) return;
        Traceback(n - l[n],i,s,l);
        s[i++] = n - l[n] ;
    }

    void Output(int s[], int l[],int b[],int n)
    {
        cout << "The optimal value is " << s[n] << endl;
        int m = 0;
        Traceback(n,m,s,l);
        s[m] = n;
        cout << "Decompose into " << m << " segments " << endl;
        for (int j = 1;j <= m;j++) {
          l[j] = l[s[j]];
```

```
        b[j] = b[s[j]];
        }
      for (int j = 1;j < = m;j + +)
        cout < < l[j] < < ' ' < < b[j] < < endl;
}
```

4. 计算复杂性

算法 Compress 显然只需 $O(n)$ 空间。由于算法 Compress 中对 j 的循环次数不超过 256，故对每一个确定的 i，可在 $O(1)$ 时间内完成 $\min\limits_{1 \leqslant j \leqslant \min\{i,256\}} \{s[i-j] + j * b\max(i-j+1, i)\}$ 的计算。因此整个算法所需的计算时间为 $O(n)$。

3.8 电路布线

在一块电路板的上、下两端分别有 n 个接线柱。根据电路设计，要求用导线 $(i, \pi(i))$ 将上端接线柱 i 与下端接线柱 $\pi(i)$ 相连，如图 3-6 所示。其中，$\pi(i), 1 \leqslant i \leqslant n$ 是 $\{1,2,\cdots,n\}$ 的一个排列。导线 $(i,\pi(i))$ 称为该电路板上的第 i 条连线。对于任何 $1 \leqslant i < j \leqslant n$，$i, j$ 两条连线相交的充分且必要条件是 $\pi(i) > \pi(j)$。

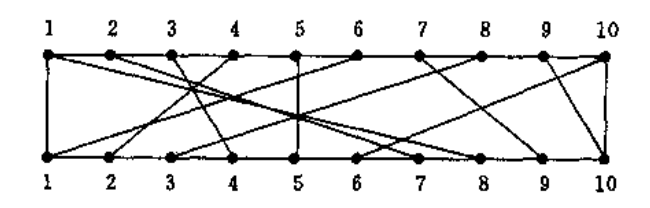


图 3-6　电路布线实例

在制作电路板时，要求将这 n 条连线分布到若干个绝缘层上。在同一层上的连线不相交。电路布线问题就是要确定将哪些连线安排在第一层上，使得该层上有尽可能多的连线。换句话说，该问题要求确定导线集 Nets $= \{(i,\pi(i)), 1 \leqslant i \leqslant n\}$ 的最大不相交子集。

1. 最优子结构性质

记 $N(i,j) = \{t \mid (t,\pi(t)) \in \text{Nets}, t \leqslant i, \pi(t) \leqslant j\}$，$N(i,j)$ 的最大不相交子集为 $\text{MNS}(i,j)$，$\text{Size}(i,j) = |\text{MNS}(i,j)|$。

(1) 当 $i = 1$ 时，

$$\text{MNS}(i,j) = N(1,j) = \begin{cases} \varnothing & j < \pi(1) \\ \{(1,\pi(1))\} & j \geqslant \pi(1) \end{cases}$$

(2) 当 $i > 1$ 时，

① $j < \pi(i)$。此时，$(i,\pi(i)) \notin N(i,j)$。故在这种情况下，$N(i,j) = N(i-1,j)$，从而，$\text{Size}(i,j) = \text{Size}(i-1,j)$。

② $j \geqslant \pi(i)$。此时，

若 $(i,\pi(i)) \in \text{MNS}(i,j)$，则对任意 $(t,\pi(t)) \in \text{MNS}(i,j)$ 有 $t < i$ 且 $\pi(t) < \pi(i)$。否则，$(t,\pi(t))$ 与 $(i,\pi(i))$ 相交。在这种情况下，$\text{MNS}(i,j) - \{(i,\pi(i))\}$ 是 $N(i-1, \pi(i)-1)$ 的一个最大不相交子集。否则子集

$$\mathrm{MNS}(i-1,\pi(i)-1)\cup\{(i,\pi(i))\}\subseteq N(i,j)$$

是比 MNS(i,j) 更大的 $N(i,j)$ 的不相交子集。这与 MNS(i,j) 的定义相矛盾

若$(i,\pi(i))\notin$ MNS(i,j),则对任意$(t,\pi(t))\in$ MNS(i,j),有 $t<i$ 从而 MNS$(i,j)\subseteq N(i-1,j)$。因此,Size$(i,j)\leqslant$ Size$(i-1,j)$。

另一方面,MNS$(i-1,j)\subseteq N(i,j)$,故又有 Size$(i,j)\geqslant$ Size$(i-1,j)$,从而 Size$(i,j)=$ Size$(i-1,j)$。

综上可知,电路布线问题满足最优子结构性质。

2. 递归计算最优值

电路布线问题的最优值为 Size(n,n). 由该问题的最优子结构性质可知:

(1) 当 $i=1$ 时,

$$\mathrm{Size}(i,j)=\begin{cases}0 & j<\pi(1)\\1 & j\geqslant\pi(1)\end{cases}$$

(2) 当 $i>1$ 时,

$$\mathrm{Size}(i,j)=\begin{cases}\mathrm{Size}(i-1,j) & j<\pi(i)\\ \max\{\mathrm{Size}(i-1,j),\mathrm{Size}(i-1,\pi(i)-1)+1\} & j\geqslant\pi(i)\end{cases}$$

据此可设计解电路布线问题的动态规划算法如下。其中用二维数组单元 size$[i][j]$ 表示函数 Size(i,j) 的值。

```
void MNS(int C[], int n, int * * size)
{
    for (int j = 0; j < C[1]; j ++)
      size[1][j] = 0;
    for (int j = C[1]; j < = n; j ++)
      size[1][j] = 1;
    for (int i = 2; i < n; i ++) {
        for (int j = 0; j < C[i]; j ++)
          size[i][j] = size[i - 1][j];
        for (int j = C[i]; j < = n; j ++)
          size[i][j] = max(size[i - 1][j], size[i - 1][C[i] - 1] + 1);
        }

    size[n][n] = max(size[n - 1][n], size[n - 1][C[n] - 1] + 1);
}


    void Traceback(int C[], int * * size, int n, int Net[], int& m)
    {
        int j = n;
        m = 0;
        for (int i = n; i > 1; i --)
            if (size[i][j] ! = size[i - 1][j]){
                Net[m ++] = i;
                j = C[i] - 1;}
        if (j > = C[1])
```

```
        Net[m++] = 1;
    }
```

3. 构造最优解

根据算法 MNS 计算出的 size[i][j] 值,容易由算法 Traceback 构造出最优解 MNS(n, n)。其中,用数组 Net[0: $m-1$] 存储 MNS(n, n) 中的 m 条连线。

4. 计算复杂性

算法 MNS 显然需要 $O(n^2)$ 计算时间和 $O(n^2)$ 空间。Traceback 需要 $O(n)$ 计算时间。

3.9 流水作业调度

n 个作业 $\{1,2,\cdots,n\}$ 要在由 2 台机器 M_1 和 M_2 组成的流水线上完成加工。每个作业加工的顺序都是先在 M_1 上加工,然后在 M_2 上加工。M_1 和 M_2 加工作业 i 所需的时间分别为 a_i 和 b_i, $1 \leqslant i \leqslant n$。流水作业调度问题要求确定这 n 个作业的最优加工顺序,使得从第一个作业在机器 M_1 上开始加工,到最后一个作业在机器 M_2 上加工完成所需的时间最少。

从直观上我们知道,一个最优调度应使机器 M_1 没有空闲时间,且机器 M_2 的空闲时间最少。在一般情况下,机器 M_2 上会有机器空闲和作业积压两种情况。

设全部作业的集合为 $N = \{1,2,\cdots,n\}$。$S \subseteq N$ 是 N 的作业子集。在一般情况下,机器 M_1 开始加工 S 中作业时,机器 M_2 还在加工其他作业,要等时间 t 后才可利用。将这种情况下完成 S 中作业所需的最短时间记为 $T(S,t)$。流水作业调度问题的最优值为 $T(N,0)$。

1. 最优子结构性质

流水作业调度问题具有最优子结构性质。

设 π 是所给 n 个流水作业的一个最优调度,它所需的加工时间为 $a_{\pi(1)} + T'$。其中,T' 是在机器 M_2 的等待时间为 $b_{\pi(1)}$ 时,安排作业 $\pi(2),\cdots,\pi(n)$ 所需的时间。

记 $S = N - \{\pi(1)\}$,则有 $T' = T(S, b_{\pi(1)})$。

事实上,由 T 的定义知 $T' \geqslant T(S, b_{\pi(1)})$。若 $T' > T(S, b_{\pi(1)})$,设 π' 是作业集 S 在机器 M_2 的等待时间为 $b_{\pi(1)}$ 情况下的一个最优调度。则 $\pi(1), \pi'(2), \cdots, \pi'(n)$ 是 N 的一个调度,且该调度所需的时间为 $a_{\pi(1)} + T(S, b_{\pi(1)}) < a_{\pi(1)} + T'$。这与 π 是 N 的一个最优调度矛盾。故 $T' \leqslant T(S, b_{\pi(1)})$。从而 $T' = T(S, b_{\pi(1)})$。这就证明了流水作业调度问题具有最优子结构的性质。

2. 递归计算最优值

由流水作业调度问题的最优子结构性质可知,

$$T(N,0) = \min_{1 \leqslant i \leqslant n} \{a_i + T(N - \{i\}, b_i)\}$$

推广到一般情形下便有

$$T(S,t) = \min_{i \in S} \{a_i + T(S - \{i\}, b_i + \max\{t - a_i, 0\})\}$$

其中,$\max\{t - a_i, 0\}$ 这一项是由于在机器 M_2 上,作业 i 须在 $\max\{t, a_i\}$ 时间之后才能开工。

因此,在机器 M_1 上完成作业 i 之后,在机器上还需

$$b_i + \max\{t, a_i\} - a_i = b_i + \max\{t - a_i, 0\}$$

时间才能完成对作业 i 的加工。

按照上述递归式,可设计出解流水作业调度问题的动态规划算法。通过对递归式的深入分析,算法还可进一步得到简化。

3. 流水作业调度的 Johnson 法则

设 π 是作业集 S 在机器 M_2 的等待时间为 t 时的任一最优调度。若在这个调度中,安排在最前面的两个作业分别是 i 和 j,即 $\pi(1) = i, \pi(2) = j$。则由动态规划递归式可得

$$T(S, t) = a_i + T(S - \{i\}, b_i + \max\{t - a_i, 0\}) = a_i + a_j + T(S - \{i, j\}, t_{ij})$$

其中,

$$\begin{aligned} t_{ij} &= b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\} \\ &= b_j + b_i - a_j + \max\{\max\{t - a_i, 0\}, 0\}, a_j - b_i\} \\ &= b_j + b_i - a_j + \max\{t - a_i, a_j - b_i, 0\} \\ &= b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_i, a_i\} \end{aligned}$$

如果作业 i 和 j 满足 $\min\{b_i, a_j\} \geqslant \min\{b_j, a_i\}$,则称作业 i 和 j 满足 Johnson 不等式。

如果作业 i 和 j 不满足 Johnson 不等式,则交换作业 i 和作业 j 的加工顺序后,作业 i 和 j 满足 Johnson 不等式。

在作业集 S 当机器 M_2 的等待时间为 t 时的调度 π 中,交换作业 i 和作业 j 的加工顺序,得到作业集 S 的另一调度 π',它所需的加工时间为

$$T'(S, t) = a_i + a_j + T(S - \{i, j\}, t_{ji})$$

其中, $t_{ji} = b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_j, a_j\}$。

当作业 i 和 j 满足 Johnson 不等式 $\min\{b_i, a_j\} \geqslant \min\{b_j, a_i\}$ 时,我们有

$$\max\{-b_i, -a_j\} \leqslant \max\{-b_j, -a_i\}$$

从而

$$a_i + a_j + \max\{-b_i, -a_j\} \leqslant a_i + a_j + \max\{-b_j, -a_i\}$$

由此可得

$$\max\{a_i + a_j - b_i, a_i\} \leqslant \max\{a_i + a_j - b_j, a_j\}$$

因此对任意 t 有

$$\max\{t, a_i + a_j - b_i, a_i\} \leqslant \max\{t, a_i + a_j - b_j, a_j\}$$

从而,$t_{ij} \leqslant t_{ji}$。由此可见 $T(S, t) \leqslant T'(S, t)$。

换句话说,当作业 i 和作业 j 不满足 Johnson 不等式时,交换它们的加工顺序后,作业 i 和 j 满足 Johnson 不等式,且不增加加工时间。由此可知,对于流水作业调度问题,必存在一个最优调度 π,使得作业 $\pi(i)$ 和 $\pi(i+1)$ 满足 Johnson 不等式

$$\min\{b_{\pi(i)}, a_{\pi(i+1)}\} \geqslant \min\{b_{\pi(i+1)}, a_{\pi(i)}\}, \quad 1 \leqslant i \leqslant n - 1$$

称这样的调度 π 为满足 Johnson 法则的调度。

进一步还可以证明,调度 π 满足 Johnson 法则当且仅当对任意 $i < j$ 有

$$\min\{b_{\pi(i)}, a_{\pi(j)}\} \geqslant \min\{b_{\pi(j)}, a_{\pi(i)}\}$$

由此可知,任意两个满足 Johnson 法则的调度具有相同的加工时间。从而所有满足 Johnson

法则的调度均为最优调度。至此,我们将流水作业调度问题转化为求满足 Johnson 法则的调度问题。

4. 算法描述

从上面的分析可知,流水作业调度问题一定存在满足 Johnson 法则的最优调度,且容易由下面的算法确定。

流水作业调度问题的 Johnson 算法:

(1) 令 $N_1 = \{i \mid a_i < b_i\}, N_2 = \{i \mid a_i \geq b_i\}$;

(2) 将 N_1 中作业依 a_i 的非减序排序;将 N_2 中作业依 b_i 的非增序排序;

(3) N_1 中作业接 N_2 中作业构成满足 Johnson 法则的最优调度。

算法可具体实现如下:

```
int FlowShop(int n, int a, int b, int c)
{
    class Jobtype{
    public:
      int operator < = (Jobtype a) const
      {return (key < = a.key);}
      int  key;
      int  index;
      bool  job;
    };

    Jobtype * d = new Jobtype [n];
    for (int i = 0; i < n; i ++) {
        d[i].key = a[i] > b[i] ? b[i]:a[i];
        d[i].job = a[i] < = b[i];
        d[i].index = i;
        }
    sort(d,n);
      int j = 0, k = n - 1;
      for (int i = 0; i < n; i ++) {
        if (d[i].job) c[j ++] = d[i].index;
        else c[k --] = d[i].index;
        }
      j = a[c[0]];
      k = j + b[c[0]];
      for (int i = 1; i < n; i ++) {
        j + = a[c[i]];
        k = j < k?k + b[c[i]]:j + b[c[i]];
        }
      delete d;
      return k;
}
```

5. 计算复杂性分析

算法 FlowShop 的主要计算时间花在对作业集的排序上。因此，在最坏情况下算法 FlowShop 所需的计算时间为 $O(n\log n)$。所需的空间显然为 $O(n)$。

3.10 0-1 背包问题

0-1背包问题：给定 n 种物品和一背包。物品 i 的重量是 w_i，其价值为 v_i，背包的容量为 c。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？

在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i。因此，该问题称为0-1 背包问题。

此问题的形式化描述是，给定 $c>0, w_i>0, v_i>0, 1\leqslant i\leqslant n$，要求找出一个 n 元0-1向量$(x_1,x_2,\cdots,x_n)$，$x_i\in\{0,1\}, 1\leqslant i\leqslant n$，使得$\sum_{i=1}^{n} w_i x_i\leqslant c$，而且$\sum_{i=1}^{n} v_i x_i$达到最大。因此，0-1背包问题是一个特殊的整数规划问题：

$$\max\sum_{i=1}^{n} v_i x_i$$

$$\begin{cases}\sum_{i=1}^{n} w_i x_i\leqslant c\\ x_i\in\{0,1\}, 1\leqslant i\leqslant n\end{cases}$$

1. 最优子结构性质

0-1背包问题具有最优子结构性质。设$(y_1,y_2,\cdots,y_n)$是所给0-1背包问题的一个最优解。则$(y_2,\cdots,y_n)$是下面相应子问题的一个最优解：

$$\max\sum_{i=2}^{n} v_i x_i$$

$$\begin{cases}\sum_{i=2}^{n} w_i x_i\leqslant c-w_1 y_1\\ x_i\in\{0,1\}, 2\leqslant i\leqslant n\end{cases}$$

因若不然，设$(z_2,\cdots,z_n)$是上述子问题的一个最优解，而$(y_2,\cdots,y_n)$不是它的最优解。由此可知，$\sum_{i=2}^{n} v_i z_i>\sum_{i=2}^{n} v_i y_i$，且 $w_1 y_1+\sum_{i=2}^{n} w_i z_i\leqslant c$。因此

$$v_1 y_1+\sum_{i=2}^{n} v_i z_i>\sum_{i=1}^{n} v_i y_i$$

$$w_1 y_1+\sum_{i=2}^{n} w_i z_i\leqslant c$$

这说明$(y_1,z_2,\cdots,z_n)$是所给0-1背包问题的一个更优解，从而$(y_1,y_2,\cdots,y_n)$不是所给0-1背包问题的最优解。此为矛盾。

2. 递归关系

设所给0-1背包问题的子问题

$$\max\sum_{k=i}^{n} v_k x_k$$

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \leqslant j \\ x_k \in \{0,1\}, i \leqslant k \leqslant n \end{cases}$$

的最优值为 $m(i,j)$,即 $m(i,j)$ 是背包容量为 j,可选择物品为 $i,i+1,\cdots,n$ 时0-1背包问题的最优值。由 0-1 背包问题的最优子结构性质,我们可以建立计算 $m(i,j)$ 的递归式如下:

$$m(i,j) = \begin{cases} \max\{m(i+1,j), m(i+1,j-w_i)+v_i\} & j \geqslant w_i \\ m(i+1,j) & 0 \leqslant j < w_i \end{cases}$$

$$m(n,j) = \begin{cases} v_n & j \geqslant w_n \\ 0 & 0 \leqslant j < w_n \end{cases}$$

3. 算法描述

基于以上讨论,当 $w_i(1 \leqslant i \leqslant n)$ 为正整数时,用二维数组 m[][] 来存储 $m(i,j)$ 的相应值,可设计解 0-1 背包问题的动态规划算法 Knapsack 如下:

```
template < class Type >
void Knapsack(Type v, int w, int c, int n, Type * * m)
{
    int jMax = min(w[n] - 1,c);
    for (int j = 0; j < = jMax; j ++)
      m[n][j] = 0;
    for (int j = w[n]; j < = c; j ++)
      m[n][j] = v[n];

    for (int i = n - 1; i > 1; i --) {
      jMax = min(w[i] - 1,c);
      for (int j = 0; j < = jMax; j ++)
        m[i][j] = m[i + 1][j];
      for (int j = w[i]; j < = c; j ++)
        m[i][j] = max(m[i + 1][j],
                      m[i + 1][j - w[i]] + v[i]);
    }
    m[1][c] = m[2][c];
    if (c > = w[1])
        m[1][c] = max(m[1][c], m[2][c - w[1]] + v[1]);
}

template < class Type >
void Traceback(Type * * m, int w, int c, int n, int x)
{
    for (int i = 1; i < n; i ++)
      if (m[i][c] = = m[i + 1][c]) x[i] = 0;
      else {x[i] = 1;
            c - = w[i];}
    x[n] = (m[n][c]) ? 1 : 0;
```

}

按上述算法 Knapsack 计算后，m[1][c]给出所要求的0-1背包问题的最优值。相应的最优解可由算法 Traceback 计算如下。如果 m[1][c] = m[2][c]，则 $x_1 = 0$，否则 $x_1 = 1$。当 $x_1 = 0$ 时，由 m[2][c] 继续构造最优解。当 $x_1 = 1$ 时，由 m[2][$c - w_1$] 继续构造最优解。依此类推，可构造出相应的最优解$(x_1, x_2, \cdots, x_n)$。

4. 计算复杂性分析

从计算 $m(i,j)$ 的递归式容易看出，上述算法 Knapsack 需要 $O(nc)$ 计算时间，而 Traceback 需要 $O(n)$ 计算时间。

上述算法 Knapsack 有两个较明显的缺点。其一是算法要求所给物品的重量 $w_i(1 \leqslant i \leqslant n)$ 是整数。其次，当背包容量 c 很大时，算法需要的计算时间较多。例如，当 $c > 2^n$ 时，算法 Knapsack 需要 $\Omega(n2^n)$ 计算时间。

事实上，注意到计算 $m(i,j)$ 的递归式在变量 j 是连续变量，即背包容量为实数时仍成立，我们可以采用以下方法克服算法 Knapsack 的上述两个缺点。

首先考察 0-1 背包问题的一个具体实例如下。

$n = 5, c = 10, w = \{2,2,6,5,4\}, v = \{6,3,5,4,6\}$。

由计算 $m(i,j)$ 的递归式，当 $i = 5$ 时，

$$m(5,j) = \begin{cases} 6 & j \geqslant 4 \\ 0 & 0 \leqslant j < 4 \end{cases}$$

该函数是关于变量 j 的阶梯状函数。由 $m(i,j)$ 的递归式容易证明，在一般情况下，对每一个确定的 $i(1 \leqslant i \leqslant n)$，函数 $m(i,j)$ 是关于变量 j 的阶梯状单调不减函数。跳跃点是这一类函数的描述特征。如函数 $m(5,j)$ 可由其两个跳跃点(0,0) 和(4,6) 惟一确定。在一般情况下，函数 $m(i,j)$ 由其全部跳跃点惟一确定。如图 3-7 所示。

在变量 j 是连续变量的情况下，我们可以对每一个确定的 $i(1 \leqslant i \leqslant n)$，用一个表 $p[i]$ 来存储函数 $m(i,j)$ 的全部跳跃点。对每一个确定的实数 j，可以通过查找表 $p[i]$ 来确定函数 $m(i,j)$ 的值。$p[i]$ 中全部跳跃点$(j, m(i,j))$ 依 j 的升序排列。由于函数 $m(i,j)$ 是关于变量 j 的阶梯状单调不减函数，故 $p[i]$ 中全部跳跃点的 $m(i,j)$ 值也是递增排列的。

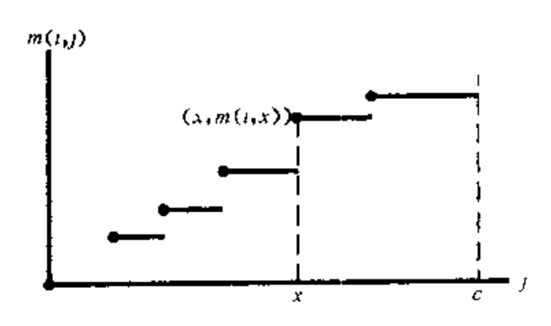


图 3-7　阶梯状单调不减函数 $m(i,j)$ 及其跳跃点

表 $p[i]$ 可依计算 $m(i,j)$ 的递归式递归地由表 $p[i+1]$ 来计算，初始时 $p[n+1] = \{(0,0)\}$。事实上，函数 $m(i,j)$ 是由函数 $m(i+1,j)$ 与函数 $m(i+1,j-w_i)+v_i$ 作 max 运算得到的。因此，函数 $m(i,j)$ 的全部跳跃点包含于函数 $m(i+1,j)$ 的跳跃点集 $p[i+1]$ 与函数 $m(i+1,j-w_i)+v_i$ 的跳跃点集 $q[i+1]$ 的并集中。易知，$(s,t) \in q[i+1]$ 当且仅当 $w_i \leqslant s \leqslant c$ 且 $(s-w_i, t-v_i) \in p[i+1]$。因此，容易由 $p[i+1]$ 确定跳跃点集 $q[i+1]$ 如下：

$$q[i+1] = p[i+1] \oplus (w_i, v_i) = \{(j+w_i, m(i,j)+v_i) \mid (j, m(i,j)) \in p[i+1]\}$$

另一方面，设(a,b) 和(c,d) 是 $p[i+1] \cup q[i+1]$ 中的两个跳跃点，则当 $c \geqslant a$ 且

$d < b$ 时，(c,d) 受控于 (a,b)，从而 (c,d) 不是 $p[i]$ 中的跳跃点。除受控跳跃点外，$p[i+1] \cup q[i+1]$ 中的其他跳跃点均为 $p[i]$ 中的跳跃点。由此可见，在递归地由表 $p[i+1]$ 计算表 $p[i]$ 时，可先由 $p[i+1]$ 计算出 $q[i+1]$，然后合并表 $p[i+1]$ 和表 $q[i+1]$，并清除其中的受控跳跃点得到表 $p[i]$。

对于上面的例子，初始时 $p[6] = \{(0,0)\}$，$(w_5, v_5) = (4,6)$。
因此有，

$$q[6] = p[6] \oplus (w_5, v_5) = \{(4,6)\}$$

由函数 $m(5,j)$ 可知，

$$p[5] = \{(0,0),(4,6)\}$$

又由 $(w_4, v_4) = (5,4)$ 知，

$$q[5] = p[5] \oplus (w_4, v_4) = \{(5,4),(9,10)\}$$

从跳跃点集 $p[5]$ 与 $q[5]$ 的并集

$$p[5] \cup q[5] = \{(0,0),(4,6),(5,4),(9,10)\}$$

中我们看到跳跃点(5,4)受控于跳跃点(4,6)。将受控跳跃点(5,4)清除后，得到 $p[4] = \{(0,0),(4,6),(9,10)\}$，从而得到函数 $m(4,j)$。

依此方式递归地计算出，

$q[4] = p[4] \oplus (6,5) = \{(6,5),(10,11)\}$

$p[3] = \{(0,0),(4,6),(9,10),(10,11)\}$

$q[3] = p[3] \oplus (2,3) = \{(2,3),(6,9)\}$

$p[2] = \{(0,0),(2,3),(4,6),(6,9),(9,10),(10,11)\}$

$q[2] = p[2] \oplus (2,6) = \{(2,6),(4,9),(6,12),(8,15)\}$

$p[1] = \{(0,0),(2,6),(4,9),(6,12),(8,15)\}$

$p[1]$ 的最后的那个跳跃点(8,15)给出所求的最优值为 $m(1,c) = 15$。

综上所述，可设计解 0-1 背包问题的改进的动态规划算法如下：

```
template < class Type >
Type Knapsack(int n, Type c, Type v[], Type w[], Type * * p, int x[])
{
    int * head = new int [n + 2];
    head[n + 1] = 0;
    p[0][0] = 0;
    p[0][1] = 0;
    int left = 0,
        right = 0,
        next = 1;
    head[n] = 1;
    for (int i = n; i > = 1; i - -) {
      int k = left;
      for (int j = left; j < = right; j + +) {
        if (p[j][0] + w[i] > c) break;
        Type y = p[j][0] + w[i],
             m = p[j][1] + v[i];
```

```
            while (k <= right && p[k][0] < y) {
              p[next][0] = p[k][0];
              p[next++][1] = p[k++][1];
              }
            if (k <= right && p[k][0] == y) {
              if (m < p[k][1]) m = p[k][1];
              k++;
              }
            if (m > p[next - 1][1]) {
              p[next][0] = y;
              p[next++][1] = m;
              }
          while (k <= right && p[k][1] <= p[next - 1][1]) k++;
          }
        while (k <= right) {
          p[next][0] = p[k][0];
          p[next++][1] = p[k++][1];
          }
        left = right + 1;
        right = next - 1;
        head[i - 1] = next;
        }
    Traceback(n,w,v,p,head,x);
    return p[next - 1][1];
}

template <class Type>
void Traceback(int n,Type w[],Type v[],Type **p,int *head,int x[])
{
    Type j = p[head[0] - 1][0],
         m = p[head[0] - 1][1];
    for (int i = 1; i <= n; i++) {
      x[i] = 0;
      for (int k = head[i + 1];k <= head[i] - 1;k++) {
        if (p[k][0] + w[i] == j && p[k][1] + v[i] == m) {
            x[i] = 1;
            j = p[k][0];
            m = p[k][1];
            break;
            }
        }
      }
}
```

上述算法的主要计算量在于计算跳跃点集 $p[i](1 \leqslant i \leqslant n)$。由于 $q[i+1] = p[i+1] \oplus (w_i, v_i)$，故计算 $q[i+1]$ 需要 $O(|p[i+1]|)$ 计算时间。合并 $p[i+1]$ 和 $q[i+1]$ 并清

除受控跳跃点也需要 $O(|p[i+1]|)$ 计算时间。从跳跃点集 $p[i]$ 的定义可以看出，$p[i]$ 中的跳跃点相应于 $x_i,\cdots,x_n$ 的 0-1 赋值。因此，$p[i]$ 中跳跃点个数不超过 2^{n-i+1}。由此可见，算法计算跳跃点集 $p[i](1\leqslant i\leqslant n)$ 所花费的计算时间为

$$O\left(\sum_{i=2}^{n}|p[i+1]|\right)=O\left(\sum_{i=2}^{n}2^{n-i}\right)=O(2^n)$$

从而，改进后算法的计算时间复杂性为 $O(2^n)$。当所给物品的重量 $w_i(1\leqslant i\leqslant n)$ 是整数时，$|p[i]|\leqslant c+1,(1\leqslant i\leqslant n)$。此时，改进后算法的计算时间复杂性为 $O(\min\{nc,2^n\})$。

3.11 最优二叉搜索树

设 $S=\{x_1,x_2,\cdots,x_n\}$ 是一个有序集，且 $x_1<x_2<\cdots<x_n$。表示有序集 S 的二叉搜索树利用二叉树的结点来存储有序集中的元素。它具有下述性质：存储于每个结点中的元素 x 大于其左子树中任一结点所存储的元素，小于其右子树中任一结点所存储的元素。二叉搜索树的叶结点是形如 (x_i,x_{i+1}) 的开区间。在表示 S 的二叉搜索树中搜索一个元素 x，返回的结果有两种情形：

(1) 在二叉搜索树的内结点中找到 $x=x_i$。

(2) 在二叉搜索树的叶结点中确定 $x\in(x_i,x_{i+1})$。

设在第(1)种情形中找到元素 $x=x_i$ 的概率为 b_i；在第(2)种情形中确定 $x\in(x_i,x_{i+1})$ 的概率为 a_i。其中约定 $x_0=-\infty,x_{n+1}=+\infty$。显然，我们有

$$a_i\geqslant 0,0\leqslant i\leqslant n;\ b_j\geqslant 0,1\leqslant j\leqslant n;\ \sum_{i=0}^{n}a_i+\sum_{j=1}^{n}b_j=1$$

$(a_0,b_1,a_1,\cdots,b_n,a_n)$ 称为集合 S 的存取概率分布。

在表示 S 的二叉搜索树 T 中，设存储元素 x_i 的结点深度为 c_i；叶结点 (x_j,x_{j+1}) 的结点深度为 d_j，则

$$p=\sum_{i=1}^{n}b_i(1+c_i)+\sum_{j=0}^{n}a_jd_j$$

表示在二叉搜索树 T 中作一次搜索所需的平均比较次数。p 又称为二叉搜索树 T 的平均路长。在一般情形下，不同的二叉搜索树的平均路长是不相同的。

最优二叉搜索树问题是对于有序集 S 及其存取概率分布 $(a_0,b_1,a_1,\cdots,b_n,a_n)$，在所有表示有序集 S 的二叉搜索树中找出一棵具有最小平均路长的二叉搜索树。

1. 最优子结构性质

二叉搜索树 T 的一棵含有结点 $x_i,\cdots,x_j$ 和叶结点 $(x_{i-1},x_i),\cdots,(x_j,x_{j+1})$ 的子树可以看作是有序集 $\{x_i,\cdots,x_j\}$ 关于全集合 $\{x_{i-1},x_{j+1}\}$ 的一棵二叉搜索树，其存取概率为下面的条件概率

$$\bar{b}_k=b_k/w_{ij},i\leqslant k\leqslant j;\ \bar{a}_h=a_h/w_{ij},i-1\leqslant h\leqslant j;$$

其中，$w_{ij}=a_{i-1}+b_i+\cdots+b_j+a_j$。

设 T_{ij} 是有序集 $\{x_i,\cdots,x_j\}$ 关于存取概率 $\{\bar{a}_{i-1},\bar{b}_i,\cdots,\bar{b}_j,\bar{a}_j\}$ 的一棵最优二叉搜索树，其平均路长为 p_{ij}。T_{ij} 的根结点存储元素 x_m。其左右子树 T_l 和 T_r 的平均路长分别为 p_l 和 p_r。由于 T_l 和 T_r 中结点深度是它们在 T_{ij} 中的结点深度减 1，故我们有

$$w_{i,j}p_{i,j} = w_{i,j} + w_{i,m-1}p_l + w_{m+1,j}p_r$$

由于 T_l 是关于集合 $\{x_i, \cdots, x_{m-1}\}$ 的一棵二叉搜索树，故 $p_l \geq p_{i,m-1}$。若 $p_l > p_{i,m-1}$，则用 $T_{i,m-1}$ 替换 T_l 可得到平均路长比 T_{ij} 更小的二叉搜索树。这与 T_{ij} 是最优二叉搜索树矛盾。故 T_l 是一棵最优二叉搜索树。同理可证 T_r 也是一棵最优二叉搜索树。因此最优二叉搜索树问题具有最优子结构性质。

2. 递归计算最优值

最优二叉搜索树 T_{ij} 的平均路长为 p_{ij}，则所求的最优值为 $p_{1,n}$。由最优二叉搜索树问题的最优子结构性质可建立计算 p_{ij} 的递归式如下

$$w_{i,j}p_{i,j} = w_{i,j} + \min_{i \leq k \leq j} \{ w_{i,k-1}p_{i,k-1} + w_{k+1,j}p_{k+1,j} \}, \ i \leq j$$

初始时，$p_{i,i-1} = 0, \ 1 \leq i \leq n$。

记 $w_{i,j}p_{i,j}$ 为 $m(i,j)$，则 $m(1,n) = w_{1,n}p_{1,n} = p_{1,n}$ 为所求的最优值。

计算 $m(i,j)$ 的递归式为

$$m(i,j) = w_{i,j} + \min_{i \leq k \leq j} \{ m(i,k-1) + m(k+1,j) \}, \ i \leq j$$

$$m(i,i-1) = 0, \ 1 \leq i \leq n$$

据此，可设计出解最优二叉搜索树问题的动态规划算法 OptimalBinarySearchTree 如下：

```
void OptimalBinarySearchTree(int a, int b, int n,
                             int * * m, int * * s, int * * w)
{
    for (int i = 0; i <= n; i++) {
        w[i + 1][i] = a[i];
        m[i + 1][i] = 0;
        }
    for (int r = 0; r < n; r++)
        for (int i = 1; i <= n - r; i++) {
          int j = i + r;
          w[i][j] = w[i][j - 1] + a[j] + b[j];
          m[i][j] = m[i + 1][j];
        s[i][j] = i;
          for (int k = i + 1; k <= j; k++) {
            int t = m[i][k - 1] + m[k + 1][j];
            if (t < m[i][j]) {
              m[i][j] = t;
             s[i][j] = k;}
            }
          m[i][j] += w[i][j];
          }
}
```

3. 构造最优解

算法 OptimalBinarySearchTree 中用 s[i][j] 保存最优子树 $T(i,j)$ 的根结点中元素。当 s[1][n] = k 时，x_k 为所求二叉搜索树根结点元素。其左子树为 $T(1,k-1)$。因此

$i = s[1][k-1]$ 表示 $T(1, k-1)$ 的根结点元素为 x_i。依此类推，容易由 s 记录的信息在 $O(n)$ 时间内构造出所求的最优二叉搜索树。

4. 计算复杂性

算法中用到 3 个二维数组 m, s 和 w, 故所需的空间为 $O(n^2)$。算法的主要计算量在于计算 $\min_{i \le k \le j}\{m(i,k-1)+m(k+1,j)\}$。对于固定的 r，它需要计算时间 $O(j-i+1)=O(r+1)$。因此算法所耗费的总时间为 $\sum_{r=0}^{n-1}\sum_{i=1}^{n-r} O(r+1) = O(n^3)$。

事实上，在上述算法中可以证明

$$\min_{i \le k \le j}\{m(i,k-1)+m(k+1,j)\} = \min_{s[i][j-1] \le k \le s[i+1][j]}\{m(i,k-1)+m(k+1,j)\}$$

由此可对算法作出进一步改进如下：

```
void OBST (int a, int b, int n,
                                  int * * m, int * * s, int * * w)
{
    for (int i = 0; i < = n; i ++) {
        w[i + 1][i] = a[i];
        m[i + 1][i] = 0;
       s[i + 1][i] = 0;
        }
    for (int r = 0; r < n; r ++)
       for (int i = 1; i < = n - r; i ++) {
          int j = i + r,
              i1 = s[i][j - 1] > i ? s[i][j - 1]:i,
              j1 = s[i + 1][j] > i ? s[i + 1][j]:j;
          w[i][j] = w[i][j - 1] + a[j] + b[j];
          m[i][j] = m[i][i1 - 1] + m[i1 + 1][j];
         s[i][j] = i1;
          for (int k = i1 + 1; k < = j1; k ++) {
              int t = m[i][k - 1] + m[k + 1][j];
              if (t < = m[i][j]) {
                  m[i][j] = t;
                s[i][j] = k;;
                }
            m[i][j] + = w[i][j];
            }
}
```

改进后算法 OBST 所需的计算时间为 $O(n^2)$，所需的空间为 $O(n^2)$。

在下一节中，我们将在较一般的意义上证明上述改进后的算法 OBST 的正确性。

3.12 动态规划加速原理

从本章前几节的讨论中我们看到，许多可用动态规划求解的问题具有类似的递归计算式。本节中，我们来考察一个常见的动态规划递归式，并讨论其计算复杂性。

设 $w(i,j) \in R$, $1 \leq i < j \leq n$。且 $m(i,j)$ 的递归计算式为

$$m(i,i) = 0,\ 1 \leq i \leq n$$

$$m(i,j) = w(i,j) + \min_{i<k\leq j}\{m(i,k-1) + m(k,j)\},\ 1 \leq i < j \leq n$$

最优二叉搜索树问题的动态规划递归式是上述递归式的特殊情形。将那里的 $w(i+1,j)$ 换成这里的 $w(i,j)$。并将那里的 $m(i+1,j)$ 换成这里的 $m(i,j)$，则得到相同的递归式。

1. $O(n^3)$ 时间算法

根据递归式，按通常方法可设计计算 $m(i,j)$ 的动态规划算法如下：

```
void DynamicProgramming(int n, int * * m, int * * s, int * * w)
{
    for (int i = 1; i <= n; i++) {
        m[i][i] = 0;
        s[i][i] = 0;
        }
    for (int r = 1; r < n; r++)
      for (int i = 1; i <= n - r; i++) {
        int j = i + r;
        w[i][j] = weight(i,j);
        m[i][j] = m[i + 1][j];
       s[i][j] = i;
        for (int k = i + 1; k < j; k++) {
            int t = m[i][k] + m[k + 1][j];
            if (t <= m[i][j]) {
                m[i][j] = t;
                s[i][j] = k;}
            }
        m[i][j] += w[i][j];
        }
}
```

算法 DynamicProgramming 需要 $O(n^3)$ 计算时间和 $O(n^2)$ 空间。

2. 四边形不等式

在上述计算 $m(i,j)$ 的递归式中，当函数 $w(i,j)$ 满足

$$w(i,j) + w(i',j') \leq w(i',j) + w(i,j'),\ i \leq i' < j \leq j'$$

时，称 w 满足四边形不等式。

当函数 $w(i,j)$ 满足 $w(i',j) \leq w(i,j')$，$i \leq i' < j \leq j'$ 时，称 W 关于区间包含关系单调。

例如，在最优二叉搜索树问题中，由 $a_i \geq 0, 0 \leq i \leq n$；$b_j \geq 0, 1 \leq j \leq n$ 知，$w(i,j)$ 显然满足单调性。另一方面，

$$w(i,j) + w(i',j') = w(i',j) + w(i,j'),\ i \leq i' < j \leq j'$$

即 w 也满足四边形不等式。

对于满足四边形不等式的单调函数 w，可推知由递归式定义的函数 $m(i,j)$ 也满足四边形不等式，即

$$m(i,j) + m(i',j') \leqslant m(i',j) + m(i,j'),\ i \leqslant i' < j \leqslant j'$$

这一性质可用数学归纳法证明。我们对四边形不等式中的“长度”$l = j' - i$ 应用数学归纳法。

当 $i = i'$ 或 $j = j'$ 时，不等式显然成立。由此可知，当 $l \leqslant 1$ 时，函数 m 满足四边形不等式。下面分两种情形进行归纳证明。

情形 1　$i < i' = j < j'$。

此时，四边形不等式简化为下面的（反）三角不等式

$$m(i,j) + m(j,j') \leqslant m(i,j')$$

设 $k = \max\{t \mid m(i,j') = m(i,t-1) + m(t,j') + w(i,j')\}$，再分两种对称情形 $k \leqslant j$ 或 $k > j$。

情形 1.1　$k \leqslant j$

此时我们有 $m(i,j') = w(i,j') + m(i,k-1) + m(k,j')$。因此，

$$\begin{aligned} m(i,j) + m(j,j') &\leqslant w(i,j) + m(i,k-1) + m(k,j) + m(j,j') \\ &\leqslant w(i,j') + m(i,k-1) + m(k,j) + m(j,j') \\ &\leqslant w(i,j') + m(i,k-1) + m(k,j') \\ &= m(i,j') \end{aligned}$$

情形 1.2　$k > j$

证明与情形 1.1 类似。

情形 2　$i < i' < j < j'$

设　$y = \max\{t \mid m(i',j) = m(i',t-1) + m(t,j) + w(i',j)\}$

　　$z = \max\{t \mid m(i,j') = m(i,t-1) + m(t,j') + w(i,j')\}$

仍需再分两种情形讨论，即 $z \leqslant y$ 或 $z > y$。我们只讨论 $z \leqslant y$ 的情形，$z > y$ 的情形是对称的。

首先注意到由 y 和 z 的定义有 $z \leqslant y \leqslant j$ 且 $i < z$。由此我们有

$$\begin{aligned} m(i,j) + m(i',j') &\leqslant w(i,j) + m(i,z-1) + m(z,j) + w(i',j') + m(i',y-1) + m(y,j') \\ &\leqslant w(i,j') + w(i',j) + m(i',y-1) + m(i,z-1) + m(z,j) + m(y,j') \\ &\leqslant w(i,j') + w(i',j) + m(i',y-1) + m(i,z-1) + m(y,j) + m(z,j') \\ &= m(i,j') + m(i'j) \end{aligned}$$

综上所述，由数学归纳法即知，$m(i,j)$ 满足四边形不等式。

定义 $s(i,j) = \max\{k \mid m(i,j) = m(i,k-1) + m(k,j) + w(i,j)\}$ 由函数 $m(i,j)$ 的四边形不等式性质可推出函数 $s(i,j)$ 的单调性，即

$$s(i,j) \leqslant s(i,j+1) \leqslant s(i+1,j+1),\ i \leqslant j$$

事实上，当 $i = j$ 时，单调性不等式显然成立。因此我们只要讨论 $i < j$ 的情形。由于对称性，我们只要证明 $s(i,j) \leqslant s(i,j+1)$。

为了便于讨论，记 $m_k(i,j) = m(i,k-1) + m(k,j) + w(i,j)$。

由 $s(i,j)$ 的定义可知，为证明 $s(i,j) \leqslant s(i,j+1)$，只要证明对所有 $i < k \leqslant k' \leqslant j$，有 $m_{k'}(i,j) \leqslant m_k(i,j)$ 蕴涵 $m_{k'}(i,j+1) \leqslant m_k(i,j+1)$。

事实上，我们可以证明一个更强的不等式

$$m_k(i,j) - m_{k'}(i,j) \leqslant m_k(i,j+1) - m_{k'}(i,j+1)$$

或等价地

$$m_k(i,j) + m_{k'}(i,j+1) \leqslant m_k(i,j+1) + m_{k'}(i,j)$$

将式中四项按它们的定义展开可得

$$m(k,j) + m(k',j+1) \leqslant m(k',j) + m(k,j+1)$$

这正是在 $k \leqslant k' \leqslant j < j+1$ 时的四边形不等式。

综上所述，可得到如下重要结论：当 w 是满足四边形不等式的单调函数时，函数 $s(i,j)$ 单调。

3. 加速算法

根据前面的讨论，当 w 是满足四边形不等式的单调函数时，函数 $s(i,j)$ 单调，从而

$$\min_{i<k\leqslant j}\{m(i,k-1) + m(k,j)\} = \min_{s(i,j-1)\leqslant k\leqslant s(i+1,j)}\{m(i,k-1) + m(k,j)\}$$

由此可对算法 DynamicProgramming 作如下改进：

```
void SpeedDynamicProgramming (int n, int * * m, int * * s, int * * w)
{
    for (int i = 1; i < = n; i ++) {
        m[i][i] = 0;
        s[i][i] = 0;
        }
    for (int r = 1; r < n; r ++)
      for (int i = 1; i < = n - r; i ++) {
        int j = i + r,
            i1 = s[i][j - 1] > i ? s[i][j - 1]:i,
            j1 = s[i + 1][j] > i ? s[i + 1][j]:j - 1;
        w[i][j] = weight(i,j);
        m[i][j] = m[i][i1] + m[i1 + 1][j];
        s[i][j] = i1;
        for (int k = i1 + 1; k < = j1; k ++) {
            int t = m[i][k] + m[k + 1][j];
            if (t < = m[i][j]) {
                m[i][j] = t;
                s[i][j] = k;}
            }
        m[i][j] + = w[i][j];
        }
}
```

改进后算法 SpeedDynamicProgramming 所需的计算时间为

$$\begin{aligned} & O\Big(\sum_{r=0}^{n-1}\sum_{i=1}^{n-r}(1 + s(i+1,i+r) - s(i,i+r-1))\Big) \\ &= O\Big(\sum_{r=0}^{n-1}(n - r + s(n-r,n) - s(1,r))\Big) \\ &= O\Big(\sum_{r=0}^{n-1} n\Big) \\ &= O(n^2) \end{aligned}$$

由于最优二叉搜索树问题的动态规划递归式中 w 是满足四边形不等式的单调函数，由前

面的讨论即知,算法 OBST 是正确的。

习题 3

3-1 设计一个 $O(n^2)$ 时间的算法,找出由 n 个数组成的序列的最长单调递增子序列。

3-2 将习题3-1 中算法的计算时间减至 $O(n\log n)$。(提示:一个长度为 i 的候选子序列的最后一个元素至少与一个长度为 $i-1$ 的候选子序列的最后一个元素一样大。通过指向输入序列中元素的指针来维持候选子序列)。

3-3 用2台处理机 A 和 B 处理 n 个作业。设第 i 个作业交给机器 A 处理时需要时间 a_i,若由机器 B 来处理,则需要时间 b_i。由于各作业的特点和机器的性能关系,很可能对于某些 i,有 $a_i \geq b_i$,而对于某些 $j, j \neq i$,有 $a_j < b_j$。既不能将一个作业分开由2台机器处理,也没有一台机器能同时处理 2 个作业。设计一个动态规划算法,使得这 2 台机器处理完这 n 个作业的时间最短(从任何一台机器开工到最后一台机器停工的总时间)。研究一个实例:$(a_1, a_2, a_3, a_4, a_5, a_6) = (2,5,7,10,5,2)$;$(b_1, b_2, b_3, b_4, b_5, b_6) = (3,8,4,11,3,4)$。

3-4 设有 n 种不同面值的硬币,各硬币的面值存于数组 T[1:n] 中。现要用这些面值的硬币来找钱。可以使用的各种面值的硬币个数不限。

(1) 当只用硬币面值 T[1],T[2],…,T[i] 时,可找出钱数 j 的最少硬币个数记为 $C(i,j)$。若只用这些硬币面值,找不出钱数 j 时,记 $C(i,j)=\infty$。给出 $C(i,j)$ 的递归表达式及其初始条件。$1 \leq i \leq n$,$1 \leq j \leq L$。

(2) 设计一个动态规划算法,对 $1 \leq j \leq L$,计算出所有的 $C(n,j)$。算法中只允许使用一个长度为 L 的数组。用 L 和 n 作为变量来表示算法的计算时间复杂性。

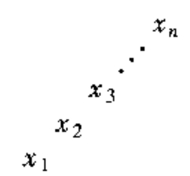
(3) 在 $C(n,j), 1 \leq j \leq L$,已计算出的情况下,设计一个贪心算法,对任意钱数 $m \leq L$,给出用最少硬币找钱 m 的方法。当 $C(n,m) \neq \infty$ 时,算法的计算时间应为 $O(n + C(n,m))$。

3-5 用关系"<"和"="将 3 个数 A, B 和 C 依序排列时,有 13 种不同的序关系:

$$A = B = C, A = B < C, A < B = C, A < B < C, A < C < B$$
$$A = C < B, B < A = C, B < A < C, B < C < A, B = C < A$$
$$C < A = B, C < A < B, C < B < A$$

若要将 n 个数依序进行排列,设计一个动态规划算法,计算出有多少种不同的序关系。要求算法只占用空间 $O(n)$,且只耗时 $O(n^2)$。

3-6 设给定 n 个变量 $x_1, x_2, \cdots, x_n$。将这些变量依序作底和各层幂,可得 n 重幂如下



这里将上述 n 重幂看作是不确定的,当在其中加入适当的括号后,才能成为一个确定的 n 重幂。不同的加括号方式导致不同的 n 重幂。例如,当 $n=4$ 时,全部4重幂有5个。试设计一个动态规划算法,对 n 个变量计算出有多少个不同的 n 重幂。要求算法只占用 $O(n)$ 空间,且只耗时 $O(n^2)$。

3-7　给定由 n 个英文单词组成的一段文章，每个单词的长度(字符个数)依序为 l_1、l_2，…，l_n。我们要在一台打印机上将这段文章"漂亮地"打印出来。打印机每行最多可打印 M 个字符。这里所说的"漂亮"的定义如下。在打印机所打印的每一行中，行首和行尾可不留空格。行中每两个单词之间留一个空格。这样，如果在一行中打印从单词 i 到单词 j 的字符，则按打印规则，应在一行中恰好打印 $\sum_{k=i}^{j} l_k + j - i$ 个字符(包括字间空格字符)，且不允许将单词打破。多余的空格数为 $M - j + i - \sum_{k=i}^{j} l_k$。除文章的最后一行外，希望每行多余的空格数尽可能少。因此，我们以各行(最后一行除外)的多余空格数的立方和达到最小作为"漂亮"的标准。试用动态规划算法设计一个"漂亮打印"方案，并分析算法的计算复杂性。

3-8　设 A 和 B 是两个字符串。我们要用最少的字符操作将字符串 A 转换为字符串 B。这里所说的字符操作包括：

(1) 删除一个字符。

(2) 插入一个字符。

(3) 将一个字符改为另一个字符。

将字符串 A 变换为字符串 B 所用的最少字符操作数称为字符串 A 到 B 的编辑距离，记为 $d(A,B)$。试设计一个有效算法，对任给的两个字符串 A 和 B，计算出它们的编辑距离 $d(A,B)$。

3-9　在一个圆形操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的两堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。试设计一个算法，计算出将 n 堆石子合并成一堆的最小得分和最大得分，并分析算法的计算复杂性。

3-10　给定一个由行数字组成的数字三角形。试设计一个算法，计算出从三角形的顶至底的一条路径，使该路径经过的数字总和最大，并分析算法的计算复杂性。

3-11　考虑下面的整数线性规划问题

$$\max \sum_{i=1}^{n} c_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} a_i x_i \leqslant b \\ x_i \text{ 为非负整数}, 1 \leqslant i \leqslant n \end{cases}$$

试设计一个解此问题的动态规划算法，并分析算法的计算复杂性。

3-12　给定 n 种物品和一背包。物品 i 的重量是 w_i，体积是 b_i，其价值为 v_i，背包的容量为 C，容积为 D。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包，不能将物品 i 装入背包多次，也不能只装入部分的物品 i。试设计一个解此问题的动态规划算法，并分析算法的计算复杂性。

3-13　考虑定义于字母表 $\sum = \{a,b,c\}$ 上的乘法表如下：

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

依此乘法表，对任一定义于$\sum$上的字符串，适当加括号后得到一个表达式。例如，对于字符串 $x = bbbba$，它的一个加括号表达式为 $(b(bb))(ba)$。依乘法表，该表达式的值为 a。试设计一个动态规划算法，对任一定义于$\sum$上的字符串 $x = x_1x_2\cdots x_n$，计算有多少种不同的加括号方式，使由 x 导出的加括号表达式的值为 a，并分析算法的计算复杂性。

3-14　Ackermann 函数 $A(m,n)$ 可递归定义如下：

$$A(m,n)=\begin{cases} n+1 & m=0 \\ A(m-1,1) & m>0, n=0 \\ A(m-1,A(m,n-1)) & m>0, n>0 \end{cases}$$

试设计一个计算 $A(m,n)$ 的动态规划算法，该算法只占用 $O(m)$ 空间（提示：用两个数组 val[0: m] 和 ind[0: m]，使得对任何 i 有 val[i] = $A(i$, ind[i]))。

3-15　长江游艇俱乐部在长江上设置了 n 个游艇出租站 $1,2,\cdots,n$。游客可在这些游艇出租站租用游艇，并在下游的任何一个游艇出租站归还游艇。游艇出租站 i 到游艇出租站 j 之间的租金为 $r(i,j)$，$1 \leqslant i < j \leqslant n$。试设计一个算法，计算出从游艇出租站 i 到游艇出租站 j 所需的最少租金，并分析算法的计算复杂性。

3-16　给定一个 $N*N$ 的方形网格，设其左上角为起点 S，坐标为 $(1,1)$，X 轴向右为正，Y 轴向下为正，每个方格边长为 1。一辆汽车从起点 S 出发驶向右下角终点 T，其坐标为 (N,N)。在若干个网格交叉点处，设置了油库，可供汽车在行驶途中加油。汽车在行驶过程中应遵守如下规则：

(1) 汽车只能沿网格边行驶，装满油后能行驶 K 条网格边。出发时汽车已装满油，在起点与终点处不设油库。

(2) 当汽车行驶经过一条网格边时，若其 X 坐标或 Y 坐标减小，则应付费用 B，否则免付费用。

(3) 汽车在行驶过程中遇油库则应加满油并付加油费用 A。

(4) 在需要时可在网格点处增设油库，并付增设油库费用 C（不含加油费用 A）。

(1) ~ (4) 中的各数 N,K,A,B,C 均为正整数。试设计一个算法，求出汽车从起点出发到达终点的一条所付费用最少的行驶路线。

3-17　给定一个 $m*n$ 的矩形网格，设其左上角为起点 S。一辆汽车从起点 S 出发驶向右下角终点 T。网格边上的数字表示距离。在若干个网格点处设置了障碍，表示该网格点不可到达。试设计一个算法，求出汽车从起点 S 出发到达终点 T 的一条行驶路程最短的路线。

3-18　关于整数的二元运算 # 定义为

$(X \# Y)$ = 十进制整数 X 的各位数字之和 * 十进制整数 Y 的最大数字 + Y 的最小数字

例如，$(9\#30) = 9*3+0 = 27$。

对于给定的十进制整数 X 和 K，由 X 和 # 运算可以组成各种不同的表达式。试设计一个算法，计算出由 X 和 # 运算组成的值为 K 的表达式最少需用多少个 # 运算。

3-19　给定一张航空图，图中的顶点表示城市，边表示城市间的直通航线。试设计一个算法，计算出一条满足下述约束条件且含城市最多的旅行路线。

(1) 从最西端的城市出发，单方向由西向东到达最东端的城市。然后，再单方向由东向西飞回起点（可途经若干城市）。

(2) 除起点城市外，每个城市最多只经过一次。

3-20　商店中每种商品都有标价。例如，一朵花的价格是2元，一个花瓶的价格是5元。为了吸引顾客，商店提供了一组优惠商品价。优惠商品是把一种或多种商品分成一组，并降价销售。例如，3朵花的价格不是6元而是5元。2个花瓶加1朵花的优惠价是10元 试设计一个算法，计算出某一顾客所购商品应付的最少费用。

3-21　一个立方体被分割成 n^3 个小立方体。每个小立方体内有一个整数。试设计一个算法，计算出所给立方体的最大子长方体。子长方体的大小由它所含所有整数之和确定。

3-22　许多操作系统采用正则表达式来实现文件匹配功能。一种简单的正则表达式由英文字母、数字及通配符“*”和“?”组成。“?”代表任意一个字符。“*”则可以代表任意多个字符。现要用正则表达式对部分文件进行操作。

(1) 试设计一个算法，找出一个正则表达式，使其能匹配的待操作文件最多，但不能匹配任何不进行操作的文件。所找出的正则表达式的长度还应是最短的。

(2) 试设计一个算法，用最少的正则表达式匹配所有待操作文件。

3-23　给定平面上 n 个点，这 n 个点的双单调欧氏旅行售货员回路是从最左点开始，严格地由左至右，然后再严格地由右向左直至出发点的闭合回路。除最左点外，该回路经过每个点恰好一次。试设计一个求这 n 个点的双单调欧氏旅行售货员回路的算法。

第4章 贪心算法

学习要点

· 理解贪心算法的概念

· 掌握贪心算法的基本要素：

(1) 最优子结构性质

(2) 贪心选择性质

· 理解贪心算法与动态规划算法的差异

· 理解贪心算法的一般理论

· 通过下面的应用范例学习贪心设计策略：

(1) 活动安排问题

(2) 最优装载问题

(3) 哈夫曼编码

(4) 单源最短路径

(5) 最小生成树

(6) 多机调度问题

当一个问题具有最优子结构性质时，我们会想到用动态规划法去解它。但有时会有更简单有效的算法。我们来看一个找硬币的例子。假设有四种硬币，它们的面值分别为二角五分、一角、五分和一分。现在要找给某顾客六角三分钱。这时，我们会不假思索地拿出2个二角五分的硬币，1个一角的硬币和3个一分的硬币交给顾客。这种找硬币方法与其他的找法相比，所拿出的硬币个数是最少的。这里，我们下意识地使用了这样的找硬币算法：首先选出一个面值不超过六角三分的最大硬币，即二角五分；然后从六角三分中减去二角五分，剩下三角八分；再选出一个面值不超过三角八分的最大硬币，即又一个二角五分，如此一直做下去。这个找硬币的方法实际上就是贪心算法。顾名思义，贪心算法总是作出在当前看来是最好的选择。也就是说贪心算法并不从整体最优上加以考虑，它所作出的选择只是在某种意义上的局部最优选择。当然，我们希望贪心算法得到的最终结果也是整体最优的。上面所说的找硬币算法得到的结果就是一个整体最优解。找硬币问题本身具有最优子结构性质，它可以用动态规划算法来解。但我们看到，用贪心算法更简单，更直接且解题效率更高。这利用了问题本身的一些特性。例如，上述找硬币的算法利用了硬币面值的特殊性。如果硬币的面值改为一分、五分和一角一分3种，而要找给顾客的是一角五分钱。还用贪心算法，我们将找给顾客1个一角一分的硬币和4个一分的硬币。然而3个五分的硬币显然是最好的找法。虽然贪心算法不是对所有问题都能得到整体最优解，但对范围相当广的许多问题它能产生整体最优解。如图的单源最短路径问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，但其最终结果却是最优解的很好的近似解。

4.1 活动安排问题

活动安排问题是可以用贪心算法有效求解的一个很好的例子。该问题要求高效地安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法，使尽可能多的活动能兼容地使用公共资源。

设有 n 个活动的集合 $E=\{1,2,\cdots,n\}$，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。每个活动 i 都有一个要求使用该资源的起始时间 s_i 和一个结束时间 f_i，且 $s_i<f_i$。如果选择了活动 i，则它在半开时间区间 $[s_i,f_i)$ 内占用资源。若区间 $[s_i,f_i)$ 与区间 $[s_j,f_j)$ 不相交，则称活动 i 与活动 j 是相容的。也就是说，当 $s_i\geqslant f_j$ 或 $s_j\geqslant f_i$ 时，活动 i 与活动 j 相容。活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合。

在下面所给出的解活动安排问题的贪心算法 GreedySelector 中，各活动的起始时间和结束时间存储于数组 s 和 f 中且按结束时间的非减序：$f_1\leqslant f_2\leqslant\cdots\leqslant f_n$ 排列。如果所给出的活动未按此序排列，我们可以用 $O(n\log n)$ 的时间将它重排。

```
template<class Type>
void GreedySelector(int n, Type s[], Type f[], bool A[])
{
  A[1] = true;
  int j = 1;
  for (int i = 2;i <= n;i++) {
    if (s[i] >= f[j]) {
      A[i] = true;
      j = i;
      }
    else A[i] = false;
    }
}
```

算法 GreedySelector 中用集合 A 来存储所选择的活动。活动 i 在集合 A 中，当且仅当 A[i]的值为 true。变量 j 用以记录最近一次加入到 A 中的活动。由于输入的活动是按其结束时间的非减序排列的，f_j 总是当前集合 A 中所有活动的最大结束时间，即

$$f_j=\max_{k\in A}\{f_k\}$$

贪心算法 GreedySelector 一开始选择活动 1，并将 j 初始化为 1。然后依次检查活动 i 是否与当前已选择的所有活动相容。若相容则将活动 i 加入到已选择活动的集合 A 中，否则不选择活动 i，而继续检查下一活动与集合 A 中活动的相容性。由于 f_j 总是当前集合 A 中所有活动的最大结束时间，故活动 i 与当前集合 A 中所有活动相容的充分且必要的条件是其开始时间 s_i 不早于最近加入集合 A 中的活动 j 的结束时间 f_j，即 $s_i\geqslant f_j$。若活动 i 与之相容，则 i 成为最近加入集合 A 中的活动，因而取代活动 j 的位置。由于输入的活动是以其完成时间的

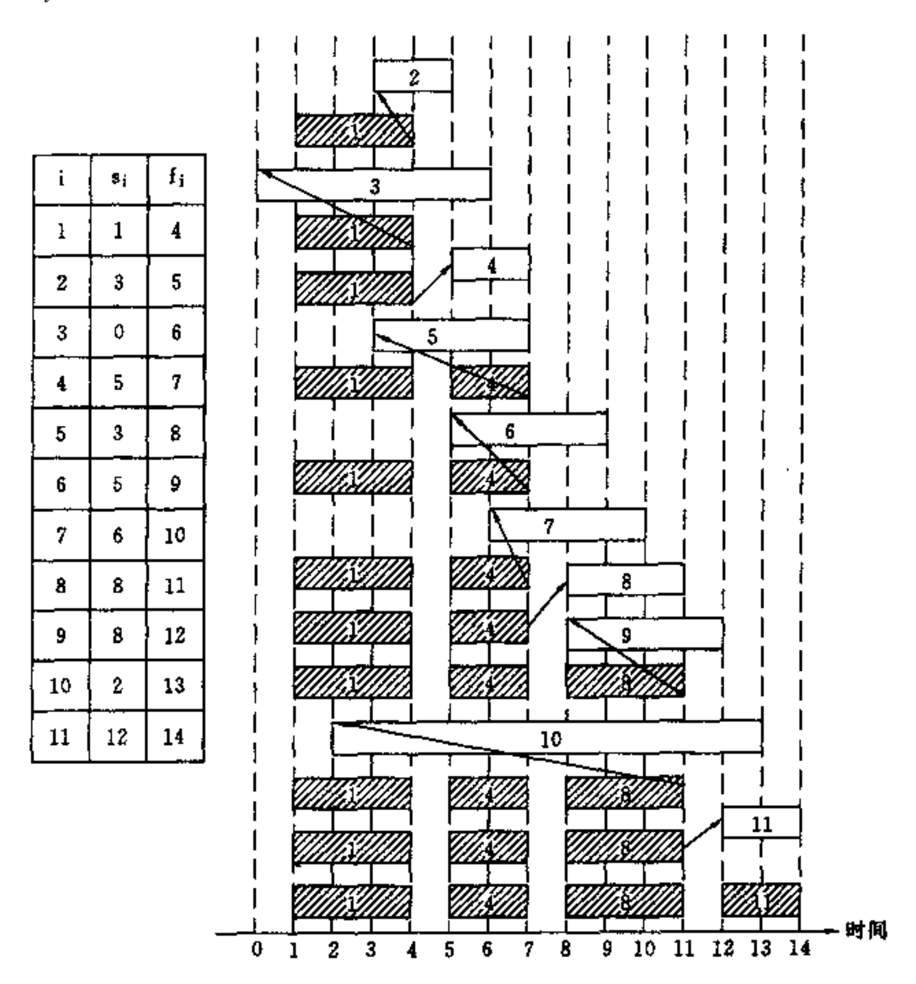
非减序排列的，所以算法 GreedySelector 每次总是选择具有最早完成时间的相容活动加入集合 A 中。直观上按这种方法选择相容活动就为未安排活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是使剩余的可安排时间段极大化，以便安排尽可能多的相容活动。

算法 GreedySelector 的效率极高。当输入的活动已按结束时间的非减序排列时，算法只需$\theta(n)$的时间来安排 n 个活动，使最多的活动能相容地使用公共资源。

例如，设待安排的 11 个活动的开始时间和结束时间按结束时间的非减序排列如下：

i	1	2	3	4	5	6	7	8	9	10	11
s[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

算法 GreedySelector 的计算过程如图 4-1 所示。



i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

图 4-1　算法 GreedySelector 的计算过程

图中每行相应于算法的一次迭代。阴影长条表示的活动是已选入集合 A 中的活动，而空白长条表示的活动是当前正在检查其相容性的活动。若被检查的活动 i 的开始时间 s_i 小于最近选择的活动 j 的结束时间 f_j，则不选择活动 i，否则选择活动 i 加入集合 A 中。

贪心算法并不总能求得问题的整体最优解。但对于活动安排问题，贪心算法 GreedySelector 却总能求得的整体最优解，即它最终所确定的相容活动集合 A 的规模最大。我们可以用数学归纳法来证明这个结论。

事实上，设 $E=\{1,2,\cdots,n\}$ 为所给的活动集合。由于 E 中活动按结束时间的非减序排列，故活动1具有最早的完成时间。首先我们要证明活动安排问题有一个最优解以贪心选择开始，即该最优解中包含活动1。设 $A\subseteq E$ 是所给的活动安排问题的一个最优解，且 A 中活动也按结束时间非减序排列，A 中的第一个活动是活动 k。若 $k=1$，则 A 就是一个以贪心选择开始的最优解。若 $k>1$，则我们设 $B=A-\{k\}\cup\{1\}$。由于 $f_1\leqslant f_k$，且 A 中活动是互为相容的，故 B 中的活动也是互为相容的。又由于 B 中活动个数与 A 中活动个数相同，且 A 是最优的，故 B 也是最优的。也就是说 B 是一个以贪心选择活动1开始的最优活动安排。因此，我们证明了总存在一个以贪心选择开始的最优活动安排方案。

进一步，在作了贪心选择，即选择了活动1后，原问题就简化为对 E 中所有与活动1相容的活动进行活动安排的子问题。即若 A 是原问题的一个最优解，则 $A'=A-\{1\}$ 是活动安排问题 $E'=\{i\in E:s_i\geqslant f_1\}$ 的一个最优解。事实上，如果我们能找到 E' 的一个解 B'，它包含比 A' 更多的活动，则将活动1加入到 B' 中将产生 E 的一个解 B，它包含比 A 更多的活动。这与 A 的最优性矛盾。因此，每一步所作的贪心选择都将问题简化为一个更小的与原问题具有相同形式的子问题。对贪心选择次数用数学归纳法即知，贪心算法 GreedySelector 最终产生原问题的一个最优解。

4.2 贪心算法的基本要素

贪心算法通过一系列的选择来得到一个问题的解。它所作的每一个选择都是当前状态下某种意义的最好选择，即贪心选择。希望通过每次所作的贪心选择导致最终结果是问题的一个最优解。这种启发式的策略并不总能奏效，然而在许多情况下确能达到预期的目的。解活动安排问题的贪心算法就是一个例子。下面我们着重讨论可以用贪心算法求解的问题的一般特征。

对于一个具体的问题，我们怎么知道是否可用贪心算法来解此问题，以及能否得到问题的一个最优解呢？这个问题很难给予肯定的回答。但是，从许多可以用贪心算法求解的问题中我们看到它们一般具有两个重要的性质：贪心选择性质和最优子结构性质。

1. 贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。在动态规划算法中，每步所作的选择往往依赖于相关子问题的解。因而只有在解出相关子问题后，才能作出选择。而在贪心算法中，仅在当前状态下作出最好选择，即局部最优选择，然后再去解作出这个选择后产生的相应的子问题。贪心算法所作的贪心选择可以依赖于以往所作过的选择，但决不依赖于将来所作的选择，也不依赖于子问题的解。正是由于这种差别，动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为一个规模更小的子问题。

对于一个具体问题，要确定它是否具有贪心选择性质，我们必须证明每一步所作的贪心选择最终导致问题的一个整体最优解。通常可以用我们在证明活动安排问题的贪心选择性质时所采用的方法来证明。首先考察问题的一个整体最优解，并证明可修改这个最优解，使其以贪

心选择开始。而且作了贪心选择后，原问题简化为一个规模更小的类似子问题。然后，用数学归纳法证明，通过每一步作贪心选择，最终可得到问题的一个整体最优解。其中，证明贪心选择后的问题简化为规模更小的类似子问题的关键在于利用该问题的最优子结构性质。

2. 最优子结构性质

当一个问题的最优解包含着它的子问题的最优解时，称此问题具有最优子结构性质。问题所具有的这个性质是该问题可用动态规划算法或贪心算法求解的一个关键特征。在活动安排问题中，其最优子结构性质表现为：若 A 是对于 E 的活动安排问题包含活动 1 的一个最优解，则相容活动集合 $A' = A - \{1\}$ 是对于 $E' = \{i \in E : s_i \geq f_1\}$ 的活动安排问题的一个最优解。

3. 贪心算法与动态规划算法的差异

贪心算法和动态规划算法都要求问题具有最优子结构性质，这是两类算法的一个共同点。但是，对于一个具有最优子结构的问题应该选用贪心算法还是动态规划算法来求解？是不是能用动态规划算法求解的问题也能用贪心算法来求解？下面我们来研究两个经典的组合优化问题，并以此来说明贪心算法与动态规划算法的主要差别。

0-1 背包问题：给定 n 种物品和一个背包。物品 i 的重量是 w_i，其价值为 v_i，背包的容量为 c。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？

在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i。

此问题的形式化描述是，给定 $c > 0, w_i > 0, v_i > 0, 1 \leq i \leq n$，要求找出一个 n 元0-1向量$(x_1, x_2, \cdots, x_n), x_i \in \{0,1\}, 1 \leq i \leq n$，使得$\sum_{i=1}^{n} w_i x_i \leq c$，而且$\sum_{i=1}^{n} v_i x_i$ 达到最大。

背包问题：与0-1 背包问题类似，所不同的是在选择物品 i 装入背包时，可以选择物品 i 的一部分，而不一定要全部装入背包。

此问题的形式化描述是，给定 $c > 0, w_i > 0, v_i > 0, 1 \leq i \leq n$，要求找出一个 n 元向量$(x_1, x_2, \cdots, x_n), 0 \leq x_i \leq 1, 1 \leq i \leq n$，使得$\sum_{i=1}^{n} w_i x_i \leq c$，而且$\sum_{i=1}^{n} v_i x_i$ 达到最大。

这两类问题都具有最优子结构性质。对于0-1 背包问题，设 A 是能够装入容量为 c 的背包的具有最大价值的物品集合，则 $A_j = A - \{j\}$ 是 $n - 1$ 个物品 $1, 2, \cdots, j-1, j+1, \cdots, n$ 可装入容量为 $c - w_j$ 的背包的具有最大价值的物品集合。对于背包问题，类似地，若它的一个最优解包含物品 j，则从该最优解中拿出所含的物品 j 的那部分重量 w，剩余的将是 $n - 1$ 个原重物品 $1, 2, \cdots, j-1, j+1, \cdots, n$ 以及重为 $w_j - w$ 的物品 j 中可装入容量为 $c - w$ 的背包且具有最大价值的物品。

虽然这两个问题极为相似，但背包问题可以用贪心算法求解，而0-1 背包问题却不能用贪心算法求解。用贪心算法解背包问题的基本步骤是，首先计算每种物品单位重量的价值 v_i / w_i，然后，依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。若将这种物品全部装入背包后，背包内的物品总重量未超过 c，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直进行下去直到背包装满为止。具体算法可描述如下：

```
void Knapsack(int n,float M,float v[],float w[],float x[])
```

```
{
  Sort(n,v,w);
  int i;
  for (i = 1;i <= n;i++) x[i] = 0;
  float c = M;
  for (i = 1;i <= n;i++) {
    if (w[i] > c) break;
    x[i] = 1;
    c -= w[i];
    }
  if (i <= n) x[i] = c/w[i];
}
```

算法 Knapsack 的主要计算时间在于将各种物品依其单位重量的价值从大到小排序。因此，算法的计算时间上界为 $O(n\log n)$。当然，为了证明算法的正确性，我们还必须证明背包问题具有贪心选择性质。

这种贪心选择策略对0-1背包问题就不适用了。看图4-2(a)中的例子，背包的容量为50千克；物品1重10千克；价值60元；物品2重20千克，价值100元；物品3重30千克，价值120元。因此，物品1每千克价值6元，物品2每千克价值5元，物品3每千克价值4元。若依贪心选择策略，应首选物品1装入背包，然而从图4-2(b)的各种情况可以看出，最优的选择方案是选择物品2和物品3装入背包。首选物品1的两种方案都不是最优的。对于背包问题，贪心选择最终可得到最优解，其选择方案如图4-2(c)所示。

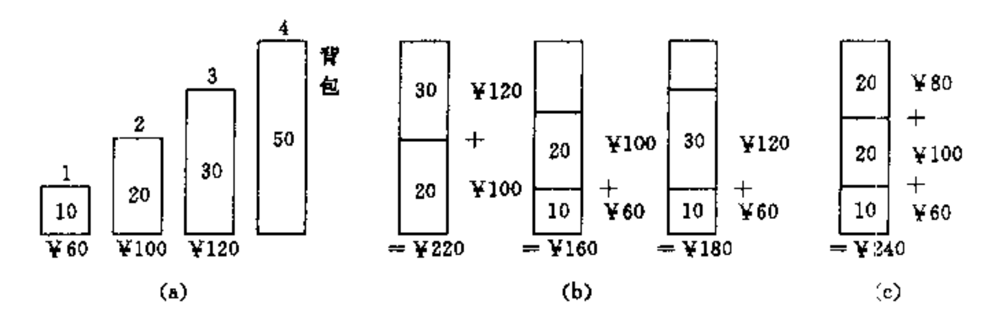


图4-2　0-1背包问题的例子

对于0-1背包问题，贪心选择之所以不能得到最优解是因为它无法保证最终能将背包装满，部分背包空间的闲置使每千克背包空间所具有的价值降低了。事实上，在考虑0-1背包问题的物品选择时，应比较选择该物品和不选择该物品所导致的最终结果，然后再作出最好选择。由此就导出许多互相重叠的子问题。这正是该问题可用动态规划算法求解的另一重要特征。动态规划算法的确可以有效地解0-1背包问题。

4.3　最优装载

有一批集装箱要装上一艘载重量为 c 的轮船。其中集装箱 i 的重量为 w_i。最优装载问题要求确定，在装载体积不受限制的情况下，应如何装载才能将尽可能多的集装箱装上轮船。该问

题可形式化描述为

$$\max \sum_{i=1}^{n} x_i$$

$$\sum_{i=1}^{n} w_i x_i \leqslant c$$

$$x_i \in \{0,1\}, 1 \leqslant i \leqslant n$$

其中,变量 $x_i = 0$ 表示不装入集装箱 i, $x_i = 1$ 表示装入集装箱 i

1. 算法描述

最优装载问题可用贪心算法求解。我们采用重量最轻者先装的贪心选择策略,由此可产生最优装载问题的一个最优解。具体算法描述如下:

```
template < class Type >
void Loading(int x[], Type w[], Type c, int n)
{
    int * t = new int [n + 1];
    Sort(w, t, n);
    for (int i = 1; i < = n; i ++)
        x[i] = 0;

    for (int i = 1; i < = n && w[t[i]] < = c; i ++) {
        x[t[i]] = 1;
        c - = w[t[i]];}
}
```

2. 贪心选择性质

设集装箱已依其重量从小到大排序,$(x_1, x_2, \cdots, x_n)$ 是最优装载问题的一个最优解。又设 $k = \min\limits_{1 \leqslant i \leqslant n} \{i \mid x_i = 1\}$。易知,如果给定的最优装载问题有解,则 $1 \leqslant k \leqslant n$。

(1) 当 $k = 1$ 时,$(x_1, x_2, \cdots, x_n)$ 是一个满足贪心选择性质的最优解。

(2) 当 $k > 1$ 时,取 $y_1 = 1; y_k = 0; y_i = x_i, 1 < i \leqslant n, i \neq k$,则

$$\sum_{i=1}^{n} w_i y_i = w_1 - w_k + \sum_{i=1}^{n} w_i x_i \leqslant \sum_{i=1}^{n} w_i x_i \leqslant c$$

因此,$(y_1, y_2, \cdots, y_n)$ 是所给最优装载问题的一个可行解。

另一方面,由 $\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} x_i$ 知,$(y_1, y_2, \cdots, y_n)$ 是一个满足贪心选择性质的最优解。所以,最优装载问题具有贪心选择性质。

3. 最优子结构性质

设 $(x_1, x_2, \cdots, x_n)$ 是最优装载问题的一个满足贪心选择性质的最优解,则易知,$x_1 = 1$,$(x_2, \cdots, x_n)$ 是轮船载重量为 $c - w_1$ 且待装船集装箱为 $\{2,3,\cdots,n\}$ 时相应最优装载问题的一

个最优解。也就是说，最优装载问题具有最优子结构性质。

由最优装载问题的贪心选择性质和最优子结构性质，容易证明算法 Loading 的正确性。

算法 Loading 的主要计算量在于将集装箱依其重量从小到大排序，故算法所需的计算时间为 $O(n\log n)$。

4.4 哈夫曼编码

哈夫曼编码是用于数据文件压缩的一个十分有效的编码方法。其压缩率通常在 20% ～ 90% 之间。哈夫曼编码算法使用一个字符在文件中出现的频率表来建立一个用 0,1 串表示各字符的最优表示方式。假设有一个数据文件包含 100 000 个字符，我们要用压缩的方式来存储它。该文件中各字符出现的频率如表4-1 所示。文件中共有 6 个不同字符出现，字符 a 出现 45 000 次，字符 b 出现 13 000 次等。

表4-1 字符出现的频率表

	a	b	c	d	e	f
频率(千次)	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

要压缩表示这个文件中的信息有多种方法。我们考虑用 0,1 码串表示字符的方法，即每个字符用惟一的一个 0,1 串来表示。若使用定长码，则表示每个不同的字符需要 3 位：a = 000, b = 001,…,f = 101。用这种方法对整个文件进行编码需要 300 000 位。我们能否做得更好些呢?使用变长码要比使用定长码好得多。通过给出现频率高的字符较短的编码，出现频率较低的字符以较长的编码，可以大大缩短总码长。表4-1 给出了一种变长码编码方案。其中，字符 a 用一位串 0 表示，而字符 f 用 4 位串 1100 表示。用这种编码方案，整个文件的总码长为：$(45\times1+13\times3+12\times3+16\times3+9\times4+5\times4)\times1\,000=224\,000$ 位。它比用定长码方案好，总码长减少约 25%。事实上，这是该文件的一个最优编码方案。

1. 前缀码

我们对每一个字符规定一个 0,1 串作为其代码，并要求任一字符的代码都不是其他字符代码的前缀。我们称这样的编码具有前缀性质，或简称为前缀码。编码的前缀性质可以使译码方法非常简单。由于任一字符的代码都不是其他字符代码的前缀，从编码文件中不断取出代表某一字符的前缀码，转换为原字符，即可逐个译出文件中的所有字符。例如表4-1 中的变长码就是一种前缀码。对于给定的 0,1 串 001011101 可惟一地分解为 0,0,101,1101，因而其译码为 aabe。

译码过程需要方便地取出编码的前缀，因此需要一个表示前缀码的合适的数据结构。为此目的，我们可以用二叉树作为前缀编码的数据结构。在表示前缀码的二叉树中，树叶代表给定的字符，并将每个字符的前缀码看作是从树根到代表该字符的树叶的一条道路。代码中每一位的 0 或 1 分别作为指示某结点到左儿子或右儿子的“路标”。例如图 4-3 中的两棵二叉树是表4-1 中两种编码方案所对应的数据结构。

容易看出，表示最优编码方案所对应的前缀码的二叉树总是一棵完全二叉树，即树中任一结点都有 2 个儿子。而定长编码方案不是最优的，其编码二叉树不是一棵完全二叉树。在一般

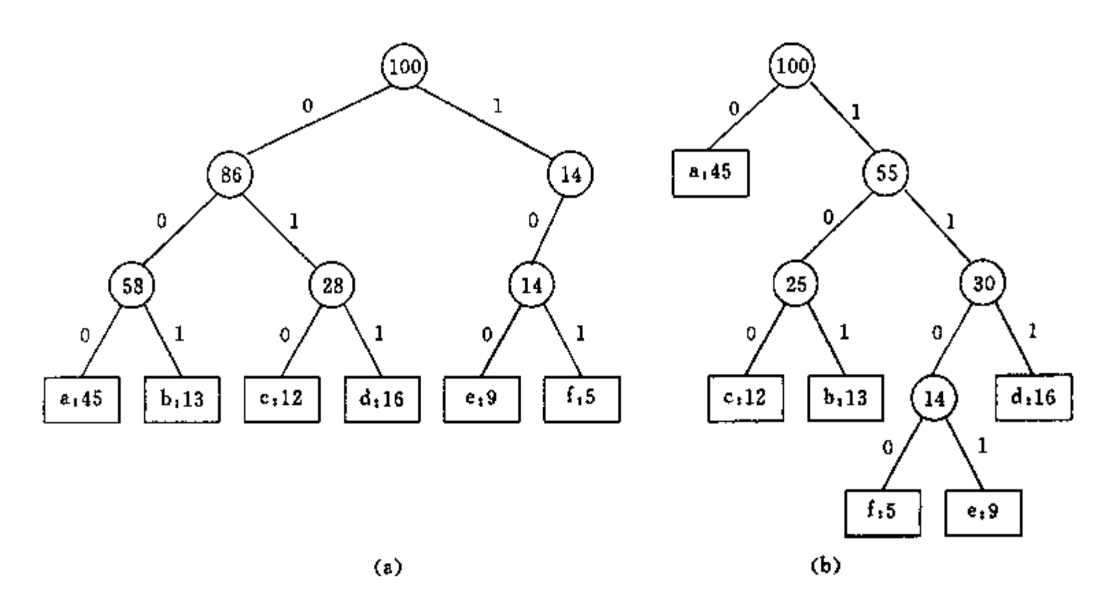


图 4-3　前缀码的二叉树表示

情况下,若 C 是编码字符集,则表示其最优前缀码的二叉树中恰有 $|C|$ 个叶子,每个叶子对应于字符集中一个字符,且该二叉树恰有 $|C|-1$ 个内部结点。

给定编码字符集 C 及其频率分布 f,即 C 中任一字符 c 以频率 $f(c)$ 在数据文件中出现。C 的一个前缀码编码方案对应于一棵二叉树 T。字符 c 在树 T 中的深度记为 $d_T(c)$。$d_T(c)$ 也是字符 c 的前缀码长。

该编码方案的平均码长定义为:$B(T) = \sum_{c \in C} f(c) d_T(c)$。

使平均码长达到最小的前缀码编码方案称为 C 的一个最优前缀码。

2. 构造哈夫曼编码

哈夫曼提出了一种构造最优前缀码的贪心算法,由此产生的编码方案称为哈夫曼算法。哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树 T。算法以 $|C|$ 个叶结点开始,执行 $|C|-1$ 次的"合并"运算后产生最终所要求的树 T。下面所给出的算法 HuffmanTree 中,编码字符集中每一字符 c 的频率是 $f(c)$。以 f 为键值的优先队列 Q 用以在作贪心选择时有效地确定算法当前要合并的两棵具有最小频率的树。一旦两棵具有最小频率的树合并后,产生一棵新的树,其频率为合并的两棵树的频率之和,并将新树插入优先队列 Q。

算法中用到的类 Huffman 定义如下:

```
template < class Type >
class Huffman {
    friend BinaryTree < int > HuffmanTree( Type [], int);
    public:
        operator Type () const {return weight;}
    private:
        BinaryTree < int > tree;
        Type weight;
};
```

算法 HuffmanTree 描述如下：

```
template < class Type >
BinaryTree < int > HuffmanTree(Type f[], int n)
{
    // 生成单结点树
    Huffman < Type > *w = new Huffman < Type > [n + 1];
    BinaryTree < int > z, zero;
    for (int i = 1; i < = n; i ++) {
        z.MakeTree(i, zero, zero);
        w[i].weight = f[i];
        w[i].tree = z;
        }

    // 建优先队列
    MinHeap < Huffman < Type > > Q(1);
    Q.Initialize(w,n,n);

    // 反复合并最小频率树
    Huffman < Type > x, y;
    for (int i = 1; i < n; i ++) {
        Q.DeleteMin(x);
        Q.DeleteMin(y);
        z.MakeTree(0, x.tree, y.tree);
        x.weight + = y.weight; x.tree = z;
        Q.Insert(x);
        }

    Q.DeleteMin(x);
    Q.Deactivate();
    delete [] w;
    return x.tree;
}
```

算法 HuffmanTree 首先用字符集 C 中每一字符 c 的频率 $f(c)$ 初始化优先队列 Q。然后不断地从优先队列 Q 中取出具有最小频率的两棵树 x 和 y，将它们合并为一棵新树 z。z 的频率是 x 和 y 的频率之和。新树 z 以 x 为其左儿子，y 为其右儿子。(也可以 y 为其左儿子，x 为其右儿子。不同的次序将产生不同的编码方案，但平均码长是相同的。) 经过 $n-1$ 次的合并后，优先队列中只剩下一棵树，即所要求的树 T。

算法 HuffmanTree 用最小堆来实现优先队列 Q。初始化优先队列需要 $O(n)$ 计算时间，由于 DeleteMin 和 Insert 只需 $O(\log n)$ 时间，$n-1$ 次的合并总共需要 $O(n\log n)$ 计算时间。因此，关于 n 个字符的哈夫曼算法的计算时间为 $O(n\log n)$。

对于表4-1 中的例子，哈夫曼算法的执行过程如图 4-4 所示。

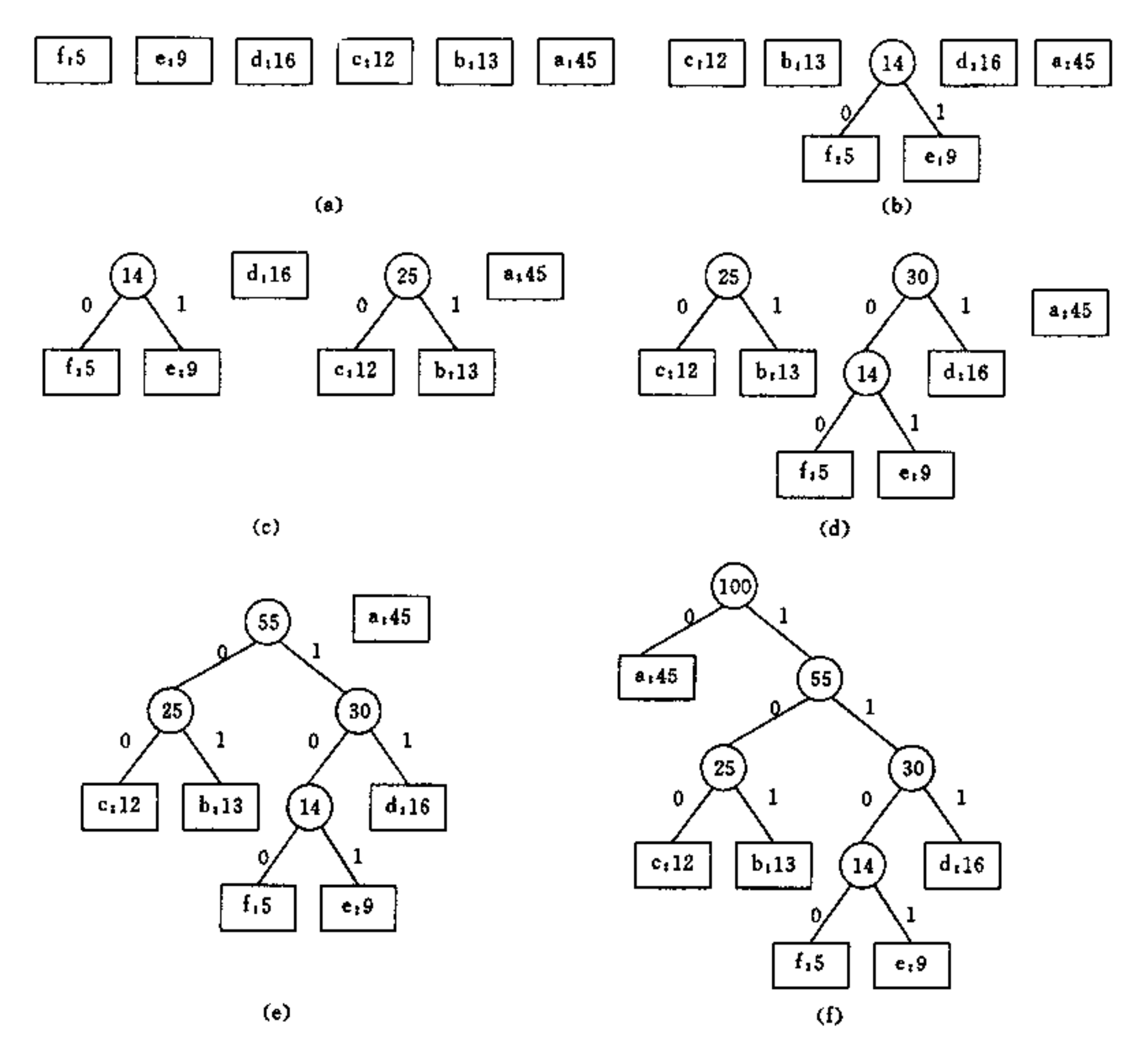


图 4-4　哈夫曼算法执行过程

由于字符集中有 6 个字符,优先队列 Q 的初始大小为 6。总共用 5 次合并得到最终的编码树 T。每次合并使优先队列 Q 的大小减 1。最终得到的树 T 即表示哈夫曼算法得到的最优前缀码——哈夫曼编码。每个字符的编码由树 T 的根到该字符的路径上各边的标号所组成,a = 0,b = 101,c = 100,d = 111,e = 1101,f = 1100。

3. 哈夫曼算法的正确性

要证明哈夫曼算法的正确性,只要证明最优前缀码问题具有贪心选择性质和最优子结构性质。

(1) 贪心选择性质

设 C 是编码字符集,C 中字符 c 的频率为 $f(c)$。又设 x 和 y 是 C 中具有最小频率的两个字符,则存在 C 的一个最优前缀码使 x 和 y 具有相同码长且仅最后一位编码不同。

证明:设二叉树 T 表示 C 的任意一个最优前缀码。我们要证明可以对 T 作适当修改后得到一棵新的二叉树 T'',使得在新树中,x 和 y 是最深叶子且为兄弟。同时新树 T'' 表示的前缀码也是 C 的一个最优前缀码。如果我们能做到这一点,则 x 和 y 在 T'' 表示的最优前缀码中就具有相同的码长且仅最后一位编码不同。

设 b 和 c 是二叉树 T 的最深叶子且为兄弟。不失一般性,可设 $f(b) \leqslant f(c)$,$f(x) \leqslant$

$f(y)$。由于 x 和 y 是 C 中具有最小频率的两个字符，故 $f(x) \leqslant f(b)$，$f(y) \leqslant f(c)$。

我们首先在树 T 中交换叶子 b 和 x 的位置得到树 T'，然后在树 T' 中再交换叶子 c 和 y 的位置，得到树 T''。如图 4-5 所示。

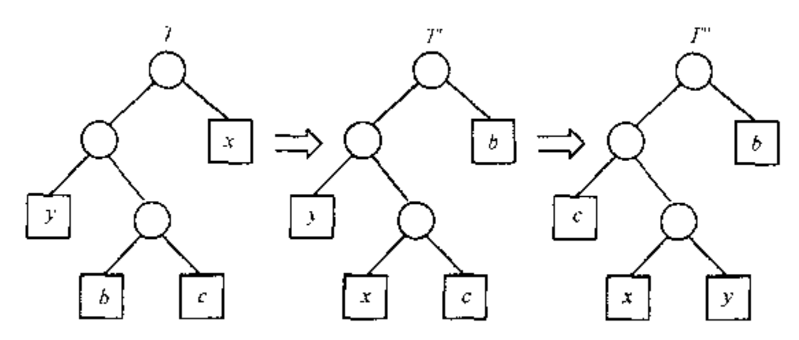


图 4-5　编码树 T 的变换

由此可知，树 T 和 T' 表示的前缀码的平均码长之差为

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x) \\
&= (f(b) - f(x))(d_T(b) - d_T(x)) \geqslant 0
\end{aligned}
$$

最后一个不等式是因为 $f(b) - f(x)$ 和 $d_T(b) - d_T(x)$ 均为非负。

类似地，可以证明在 T' 中交换 y 与 c 的位置也不增加平均码长，即 $B(T') - B(T'')$ 也是非负的。由此可知 $B(T'') \leqslant B(T') \leqslant B(T)$。另一方面，由于 T 所表示的前缀码是最优的，故 $B(T) \leqslant B(T'')$。因此，$B(T) = B(T'')$，即 T'' 表示的前缀码也是最优前缀码，且 x 和 y 具有最长的码长，同时仅最后一位编码不同。

(2) 最优子结构性质

设 T 是表示字符集 C 的一个最优前缀码的完全二叉树。C 中字符 c 的出现频率为 $f(c)$。设 x 和 y 是树 T 中的两个叶子且为兄弟，z 是它们的父亲。若将 z 看作是具有频率 $f(z) = f(x) + f(y)$ 的字符，则树 $T' = T - \{x, y\}$ 表示字符集 $C' = C - \{x, y\} \cup \{z\}$ 的一个最优前缀码。

证明：我们首先证明 T 的平均码长 $B(T)$ 可用 T' 的平均码长 $B(T')$ 来表示。

事实上，对任意 $c \in C - \{x, y\}$ 有 $d_T(c) = d_{T'}(c)$，故 $f(c) d_T(c) = f(c) d_{T'}(c)$。

另一方面，$d_T(x) = d_T(y) = d_{T'}(z) + 1$，故

$$
\begin{aligned}
f(x) d_T(x) + f(y) d_T(y) &= (f(x) + f(y))(d_{T'}(z) + 1) \\
&= f(x) + f(y) + f(z) d_{T'}(z)
\end{aligned}
$$

由此即知，$B(T) = B(T') + f(x) + f(y)$。

若 T' 所表示的字符集 C' 的前缀码不是最优的，则有 T'' 表示的 C' 的前缀码使得 $B(T'') < B(T')$。由于 z 被看作是 C' 中的一个字符，故 z 在 T'' 中是一树叶。若将 x 和 y 加入树 T'' 中作为 z 的儿子，则得到表示字符集 C 的前缀码的二叉树 T'''，且有

$$
\begin{aligned}
B(T''') &= B(T'') + f(x) + f(y) \\
&< B(T') + f(x) + f(y) = B(T)
\end{aligned}
$$

这与 T 的最优性矛盾。故 T' 所表示的 C' 的前缀码是最优的。

由贪心选择性质和最优子结构性质立即可推出：哈夫曼算法是正确的，即 HuffmanTree 产

生 C 的一棵最优前缀编码树。

4.5 单源最短路径

给定一个带权有向图 $G = (V, E)$，其中每条边的权是一个非负实数。另外，还给定 V 中的一个顶点，称为源。现在我们要计算从源到所有其他各顶点的最短路长度。这里路的长度是指路上各边权之和。这个问题通常称为单源最短路径问题。

1. 算法基本思想

Dijkstra算法是解单源最短路径问题的一个贪心算法。其基本思想是，设置一个顶点集合 S 并不断地作贪心选择来扩充这个集合。一个顶点属于集合 S 当且仅当从源到该顶点的最短路径长度已知。初始时，S 中仅含有源。设 u 是 G 的某一个顶点，我们把从源到 u 且中间只经过 S 中顶点的路称为从源到 u 的特殊路径，并用数组 dist 来记录当前每个顶点所对应的最短特殊路径长度。Dijkstra算法每次从 $V - S$ 中取出具有最短特殊路长度的顶点 u，将 u 添加到 S 中，同时对数组 dist 作必要的修改。一旦 S 包含了所有 V 中顶点，dist 就记录了从源到所有其他顶点之间的最短路径长度。

Dijkstra算法可描述如下，其中输入的带权有向图是 $G = (V, E)$，$V = \{1, 2, \cdots, n\}$，顶点 v 是源。c是一个二维数组，c[i][j] 表示边 (i, j) 的权。当 $(i, j) \notin E$ 时，c[i][j] 是一个大数。dist[i] 表示当前从源到顶点 i 的最短特殊路径长度。

```
template < class Type >
void Dijkstra(int n,int v,Type dist[],int prev[],Type * * c)
{// 单源最短路径问题的 Dijkstra 算法
  bool s[maxint];
  for (int i = 1; i < = n; i ++){
    dist[i] = c[v][i];
    s[i] = false;
    if (dist[i] = = maxint) prev[i] = 0;
    else prev[i] = v;
    }
  dist[v] = 0; s[v] = true;
  for (int i = 1; i < n; i ++){
    int temp = maxint;
    int u = v;
    for (int j = 1;j < = n;j ++)
      if ((!s[j])&&(dist[j] < temp)) {
        u = j;
        temp = dist[j];
        }
    s[u] = true;
    for (int j = 1;j < = n;j ++)
      if ((!s[j])&&(c[u][j] < maxint)) {
```

```
            Type newdist = dist[u] + c[u][j];
          if (newdist < dist[j])
            dist[j] = newdist;
            prev[j] = u;
            }
          }
      }
    }
```

例如,对图4-6中的有向图,应用Dijkstra算法计算从源顶点1到其他顶点间最短路径的过程列在表4-2中。

表4-2 Dijkstra算法的迭代过程

迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	maxint	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

上述Dijkstra算法只求出从源顶点到其他顶点间的最短路径长度。如果还要求出相应的最短路径,可以用算法中数组prev记录的信息求出相应的最短路径。算法中数组prev[i]记录的是从源到顶点i的最短路径上i的前一个顶点。初始时,对所有$i \neq 1$,置prev[i] = v。在Dijkstra算法中更新最短路径长度时,只要 dist[u] + c[u][i] < dist[i]时,就置 prev[i] = u。当Dijkstra算法终止时,就可以根据数组prev找到从源到i的最短路径上每个顶点的前一个顶点,从而找到从源到i的最短路径。

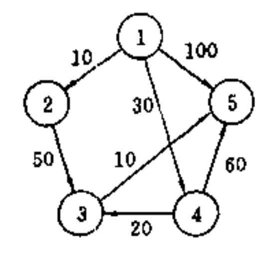


图4-6 一个带权有向图

例如,对于图4-6中的有向图,经Dijkstra算法计算后可得数组prev具有值prev[2] = 1, prev[3] = 4, prev[4] = 1, prev[5] = 3。如果要找出顶点1到顶点5的最短路径,可以从数组prev得到顶点5的前一个顶点是3,3的前一个顶点是4,4的前一个顶点是1。于是从顶点1到顶点5的最短路径是1,4,3,5。

2. 算法的正确性和计算复杂性

下面我们来讨论Dijkstra算法的正确性和计算复杂性。

(1) 贪心选择性质

Dijkstra算法是应用贪心算法设计策略的又一个典型例子。它所作的贪心选择是从$V - S$中选择具有最短特殊路径的顶点u,从而确定从源到u的最短路径长度dist[u]。这种贪心选择为什么能导致最优解呢?换句话说,为什么从源到u没有更短的其他路径呢?事实上,如果存在一条从源到u且长度比dist[u]更短的路,设这条路初次走出S之外到达的顶点为$x \in V - S$,然后徘徊于S内外若干次,最后离开S到达u。如图4-7所示。

在这条路径上,分别记$d(v, x)$, $d(x, u)$和$d(v, u)$为顶点v到顶点x,顶点x到顶点u和顶点v到顶点u的路长,那么,我们有

$$\text{dist}[x] \leqslant d(v,x)$$
$$d(v,x) + d(x,u) = d(v,u) < \text{dist}[u]$$

利用边权的非负性,可知 $d(x,u) \geqslant 0$,从而推得 $\text{dist}[x] < \text{dist}[u]$。此为矛盾。这就证明了 dist[$u$]是从源到顶点 u 的最短路径长度。

(2) 最优子结构性质

要完成 Dijkstra 算法正确性的证明,我们还必须证明最优子结构性质,即算法中确定的 dist[u]确实是当前从源到顶点 u 的最短特殊路径长度。为此,我们只要考察算法在添加 u 到 S 中后,dist[u]的值所起的变化就行了。我们将添加 u 之前的 S 称为老的 S。当添加了 u 之后,可能出现一条到顶点 i 的新的特殊路。如果这条新特殊路是先经过老的 S 到达顶点 u,然后从 u 经一条边直接到达顶点 i,则这种路的最短的长度是 $\text{dist}[u] + c[u][i]$。这时,如果 $\text{dist}[u] + c[u][i] < \text{dist}[i]$,则算法中用 $\text{dist}[u] + c[u][i]$ 作为 dist[i] 的新值。如果这条新特殊路径经过老的 S 到达 u 后,不是从 u 经一条边直接到达 i,而是像图 4-8 那样,回到老的 S 中某个顶点 x,最后才到达顶点 i,那么由于 x 在老的 S 中,因此 x 比 u 先加入 S,故图 4-8 中从源到 x 的路的长度比从源到 u,再从 u 到 x 的路的长度小。于是当前 dist[i] 的值小于图 4-8 中从源经 x 到 i 的路的长度,也小于图中从源经 u 和 x,最后到达 i 的路的长度。因此,我们在算法中不必考虑这种路。由此即知,不论算法中 dist[u] 的值是否有变化,它总是关于当前顶点集 S 到顶点 u 的最短特殊路径长度。

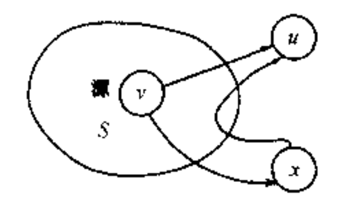


图 4-7　从源到 u 的最短路径

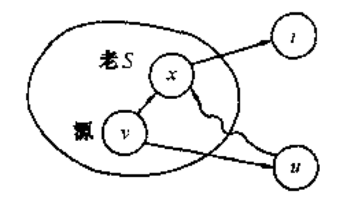


图 4-8　非最短的特殊路径

(3) 计算复杂性

对于一个具有 n 个顶点和 e 条边的带权有向图,如果用带权邻接矩阵表示这个图,那么 Dijkstra 算法的主循环体需要 $O(n)$ 时间。这个循环需要执行 $n-1$ 次,所以完成循环需要 $O(n^2)$ 时间。算法的其余部分所需要时间不超过 $O(n^2)$。

4.6　最小生成树

设 $G = (V,E)$ 是一个无向连通带权图,即一个网络。E 中每条边 (v,w) 的权为 $c[v][w]$。如果 G 的一个子图 G' 是一棵包含 G 的所有顶点的树,则称 G' 为 G 的生成树。生成树上各边权的总和称为该生成树的耗费。在 G 的所有生成树中,耗费最小的生成树称为 G 的最小生成树。

网络的最小生成树在实际中有广泛应用。例如,在设计通信网络时,用图的顶点表示城市,用边 (v,w) 的权 $c[v][w]$ 表示建立城市 v 和城市 w 之间的通信线路所需的费用,则最小生成树就给出了建立通信网络的最经济的方案。

1. 最小生成树的性质

用贪心算法设计策略可以设计出构造最小生成树的有效算法。本节中要介绍的构造最小

生成树的 Prim 算法和 Kruskal 算法都可以看作是应用贪心算法设计策略的典型例子。尽管这两个算法做贪心选择的方式不同，但它们都利用了下面的最小生成树性质：

设 $G=(V,E)$ 是一个连通带权图，U 是 V 的一个真子集。如果 $(u,v)\in E$，且 $u\in U$，$v\in V-U$，且在所有这样的边中，(u,v) 的权 $c[u][v]$ 最小，那么一定存在 G 的一棵最小生成树，它以 (u,v) 为其中一条边。这个性质有时也称为 MST 性质。MST 性质可证明如下。

假设 G 的任何一棵最小生成树都不含边 (u,v)。将边 (u,v) 添加到 G 的一棵最小生成树 T 上，将产生一个含有边 (u,v) 的圈，并且在这个圈上有一条不同于 (u,v) 的边 (u',v')，使得 $u'\in U$，$v'\in V-U$，如图 4-9 所示。

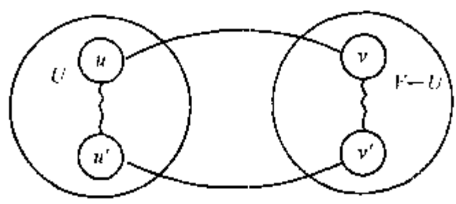


图 4-9 含边 (u,v) 的圈

将边 (u',v') 删去，得到 G 的另一棵生成树 T'。由于 $c[u][v]\leqslant c[u'][v']$，所以 T' 的耗费 $\leqslant T$ 的耗费。于是 T' 是一棵含有边 (u,v) 的最小生成树，这与假设矛盾。

2. Prim 算法

设 $G=(V,E)$ 是一个连通带权图，$V=\{1,2,\cdots,n\}$。构造 G 的一棵最小生成树的 Prim 算法的基本思想是：首先置 $S=\{1\}$，然后，只要 S 是 V 的真子集，就作如下的贪心选择：选取满足条件 $i\in S$，$j\in V-S$，且 $c[i][j]$ 最小的边，并将顶点 j 添加到 S 中。这个过程一直进行到 $S=V$ 时为止。在这个过程中选取到的所有边恰好构成 G 的一棵最小生成树。算法描述如下：

```
void Prim(int n,Type * * c)
{
  T = ∅;
  S = {1};
  while (S!= V) {
    (i,j) = i∈S 且 j∈V - S 的最小权边;
    T = T ∪ {(i,j)};
    S = S ∪ {j};
    }
}
```

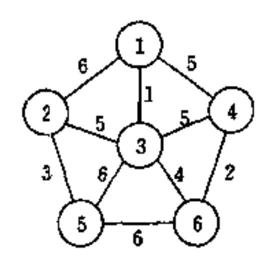


图 4-10 连通带权图

算法结束时，T 中包含 G 的 $n-1$ 条边。利用最小生成树性质和数学归纳法容易证明，上述算法中的边集合 T 始终包含 G 的某棵最小生成树中的边。因此，在算法结束时，T 中的所有边构成 G 的一棵最小生成树。

例如，对于图 4-10 中的带权图，按 Prim 算法选取边的过程如图 4-11 所示。

在上述 Prim 算法中，我们还应当考虑如何有效地找出满足条件 $i\in S$，$j\in V-S$，且权 $c[i][j]$ 最小的边 (i,j)。实现这个目的

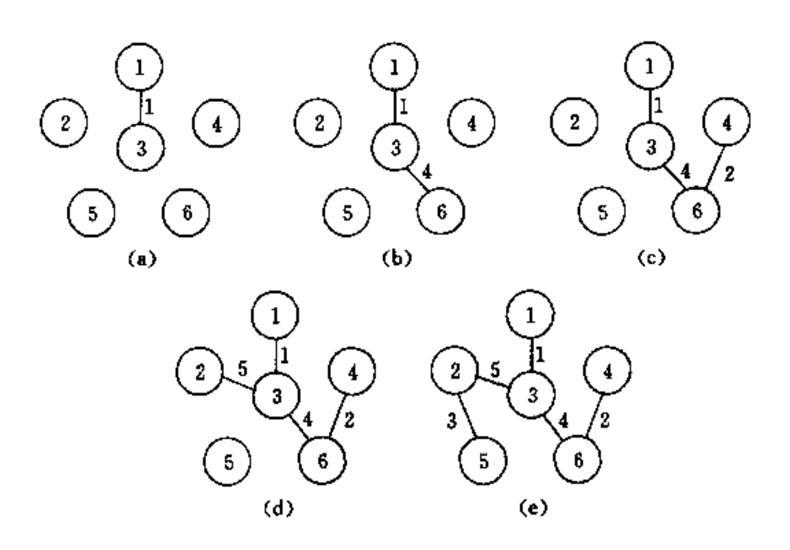


图 4-11　Prim 算法选边过程

的一个较简单的办法是设置两个数组 closest 和 lowcost。对于每一个 $j \in V - S$，closest[j] 是 j 在 S 中的一个邻接顶点，它与 j 在 S 中的其他邻接顶点 k 相比较有 $c[j][\text{closest}[j]] \leqslant c[j][k]$。lowcost[$j$] 的值就是 $c[j][\text{closest}[j]]$。

在 Prim 算法执行过程中，先找出 $V - S$ 中使 lowcost 值最小的顶点 j，然后根据数组 closest 选取边 (j, closest[j])，最后将 j 添加到 S 中，并对 closest 和 lowcost 作必要的修改。

用这个办法实现的 Prim 算法可描述如下，其中，c 是一个二维数组，c[i][j] 表示边 (i, j) 的权。

```
template < class Type >
void Prim(int n, Type * * c)
{
    Type lowcost[maxint];
    int closest[maxint];
    bool s[maxint];

    s[1] = true;
    for (int i = 2; i <= n; i++){
      lowcost[i] = c[1][i];
      closest[i] = 1;
      s[i] = false;
      }
    for (int i = 1; i < n; i++){
      Type min = inf;
      int j = 1;
      for (int k = 2; k <= n; k++)
        if ((lowcost[k] < min)&&(!s[k])) {
          min = lowcost[k];
          j = k;
          }
      cout << j << ' ' << closest[j] << endl;
```

```
          s[j] = true;
          for (int k = 2; k <= n; k++)
            if ((c[j][k] < lowcost[k]) && (!s[k])) {
              lowcost[k] = c[j][k];
              closest[k] = j;
              }
        }
  }
```

易知,上述算法 Prim 所需的计算时间为 $O(n^2)$

3. Kruskal 算法

构造最小生成树的另一个常用算法是 Kruskal 算法。当图的边数为 e 时,Kruskal 算法所需的时间是 $O(e\log e)$。当 $e = \Omega(n^2)$ 时,Kruskal算法比 Prim算法差,但当 $e = o(n^2)$ 时,Kruskal 算法却比 Prim 算法好得多。

给定无向连通带权图 $G = (V, E)$, $V = \{1,2,\cdots,n\}$。Kruskal算法构造 G 的最小生成树的基本思想是:首先将 G 的 n 个顶点看成 n 个孤立的连通分支,将所有的边按权从小到大排序。然后从第一条边开始,依边权递增的顺序查看每一条边,并按下述方法连接两个不同的连通分支:当查看到第 k 条边(v, w)时,如果端点 v 和 w 分别是当前两个不同的连通分支 $T1$ 和 $T2$ 中的顶点时,就用边(v, w)将 $T1$ 和 $T2$ 连接成一个连通分支,然后继续查看第 $k+1$ 条边;如果端点 v 和 w 在当前的同一个连通分支中,就直接再查看第 $k+1$ 条边。这个过程一直进行到只剩下一个连通分支时为止。此时,这个连通分支就是 G 的一棵最小生成树。

例如,对图 4-10 中的连通带权图,按 Kruskal算法顺序得到的最小生成树上的边如图 4-12 所示。

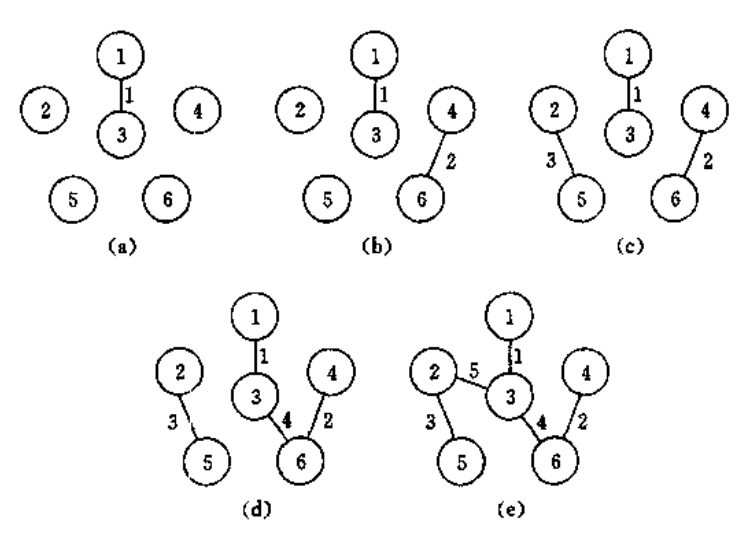


图 4-12 Kruskal 算法选边过程

关于集合的一些基本运算可用于实现 Kruskal 算法。Kruskal 算法中按权的递增顺序查看的边的序列可以看作是一个优先队列,它的优先级即为边权。顺序查看就是对这个优先队列执行 DeleteMin 运算。我们可以用堆来实现这个优先队列。

另外,在 Kruskal算法中,还要对一个由连通分支组成的集合不断进行修改。将这个由连通

分支组成的集合记为 U,则需要用到的集合的基本运算有:

(1) Union(a,b):将 U 中两个连通分支 a 和 b 连接起来,所得的结果称为 A 或 B。

(2) Find(v):返回 U 中包含顶点 v 的连通分支的名字。这个运算用来确定某条边的两个端点所属的连通分支。

这些基本运算实际上是抽象数据类型并查集 UnionFind 所支持的基本运算。

利用优先队列和并查集这两个抽象数据类型可实现 Kruskal 算法如下:

```
template < class Type >
class EdgeNode {
    friend ostream& operator < < (ostream&, EdgeNode < Type > );
    friend bool Kruskal(int,int,EdgeNode < Type > *,EdgeNode < Type > * );
    friend void main(void);
    public:
        operator Type () const {return weight;}
    private:
        Type weight;
        int u, v;
};
```

```
template < class Type >
bool Kruskal(int n,int e,EdgeNode < Type > E[],EdgeNode < Type > t[])
{
        MinHeap < EdgeNode < Type > > H(1);
        H.Initialize(E, e, e);

        UnionFind U(n);

        int k = 0;
        while (e && k < n - 1) {
            EdgeNode < int > x;
            H.DeleteMin(x);
            e --;
            int a = U.Find(x.u);
            int b = U.Find(x.v);
            if (a ! = b) {
                t[k ++] = x;
                U.Union(a,b);}
            }

        H.Deactivate();
        return (k = = n - 1);
```

设输入的连通带权图有 e 条边，则将这些边依其权组成优先队列需要 $O(e)$ 时间。在上述算法的 while 循环中，DeleteMin 运算需要 $O(\log e)$ 时间，因此关于优先队列所作运算的时间为 $O(e\log e)$。实现 UnionFind 所需的时间为 $O(e\log e)$ 或 $O(e\log^* e)$。所以 Kruskal 算法所需的计算时间为 $O(e\log e)$。

4.7 多机调度问题

设有 n 个独立的作业 $\{1,2,\cdots,n\}$，由 m 台相同的机器进行加工处理。作业 i 所需的处理时间为 t_i。现约定，任何作业可以在任何一台机器上加工处理，但未完工前不允许中断处理。任何作业不能拆分成更小的子作业。

多机调度问题要求给出一种作业调度方案，使所给的 n 个作业在尽可能短的时间内由 m 台机器加工处理完成。

这个问题是一个 NP 完全问题，到目前为止还没有一个有效的解法。对于这一类问题，用贪心选择策略有时可以设计出较好的近似算法。采用最长处理时间作业优先的贪心选择策略可以设计出解多机调度问题的较好的近似算法。按此策略，当 $n \leq m$ 时，只要将机器 i 的 $[0, t_i]$ 时间区间分配给作业 i 即可。

当 $n > m$ 时，首先将 n 个作业依其所需的处理时间从大到小排序。然后依此顺序将作业分配给空闲的处理机。

实现该策略的贪心算法 Greedy 可描述如下：

```
class JobNode {
    friend void Greedy(JobNode *, int, int);
    friend void main(void);
    public:
        operator int () const {return time;}
    private:
        int ID,
            time;
};

class MachineNode {
    friend void Greedy(JobNode *, int, int);
    public:
        operator int () const {return avail;}
    private:
        int ID,
            avail;
};
```

```
template < class Type >
void Greedy(Type a[], int n, int m)
{
    if (n < = m) {
        cout < < "为每个作业分配一台机器." < < endl;
        return; }
    Sort(a,n);
    MinHeap < MachineNode > H(m);
    MachineNode x;
    for (int i = 1; i < = m; i++) {
        x.avail = 0;
        x.ID = i;
        H.Insert(x);
        }
    for (int i = n; i > = 1; i--) {
        H.DeleteMin(x);
        cout < < "将机器" < < x.ID < < "从"
            < < x.avail < < "到"
            < < (x.avail + a[i].time)
            < < "的时间段分配给作业" < < a[i].ID < < endl;
        x.avail + = a[i].time;
        H.Insert(x);
        }
}
```

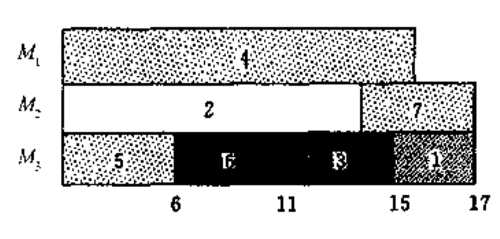


图 4-13　多机调度示例

当 $n \leqslant m$ 时算法 Greedy 需要 $O(1)$ 时间。

当 $n > m$ 时，排序耗时 $O(n\log n)$。初始化堆需要 $O(m)$ 时间。关于堆的 DeleteMin 和 Insert 运算共耗时 $O(n\log m)$，因此算法 Greedy 所需的计算时间为

$$O(n\log n + n\log m) = O(n\log n)$$

例如，设 7 个独立作业{1,2,3,4,5,6,7}由 3 台机器 M_1，M_2 和 M_3 来加工处理。各作业所需的处理时间分别为{2,14,4,16,6,5,3}。按算法 Greedy 产生的作业调度如图 4-13 所示，所需的加工时间为 17。

4.8　贪心算法的理论基础

借助于一个称为“拟阵”的工具，我们可以建立一个关于贪心算法的较一般的理论。这个理论在确定何时使用贪心算法可以得到问题的整体最优解时十分有用。

1. 拟阵

一个拟阵 M 定义为满足下面三个条件的有序对(S,I):

(1) S 是一个非空有限集.

(2) I 是 S 的一类具有遗传性质的独立子集族,即若 $B \in I$,则 B 是 S 的独立子集,且 B 的任意子集也都是 S 的独立子集。空集$\varnothing$必为 I 的一个成员。

(3) I 满足交换性质,即若 $A \in I, B \in I$ 且 $|A| < |B|$,则存在某一元素 $x \in B - A$,使得 $A \cup \{x\} \in I$。

例如,设 S 是一给定矩阵中行向量的集合,I 是 S 的线性独立子集族,则由线性空间理论容易证明(S,I)是一拟阵。

拟阵的另一个例子是无向图 $G = (V,E)$ 的图拟阵 $M_G = (S_G, I_G)$。其中,S_G 定义为图 G 的边集 E。I_G 定义为 S_G 的无循环边集族。即 $A \in I_G$ 当且仅当它在图 G 中构成一个森林、

依此定义,$M_G = (S_G, I_G)$ 是一个拟阵。事实上,$S_G = E$ 是一个有限集。由于从 S_G 的一个无循环边集中去掉若干边不会产生循环,即森林的任一子集还是森林,因此 I_G 具有遗传性质。进一步,我们还可证明 I_G 满足交换性质。设 A 和 B 是图 G 的两个森林且 $|B| > |A|$,即 A 和 B 都是无循环边集,且 B 中的边数比 A 多。由于图 G 中有 k 条边的森林恰由 $|V| - k$ 棵树组成。(从 G 中 $|V|$ 个顶点组成的森林开始,每增加一条边就减少一棵树。)因此森林 B 中的树比森林 A 中的树少。由此可推出森林 B 中存在一棵树 T,它的顶点在森林 A 的不同的两棵树中。又由于树 T 是连通的,故 T 中必有一边(u,v)使得顶点 u 和 v 在森林 A 的不同的两棵树中。将此边(u,v)加入森林 A 不会产生循环。因此,I_G 满足交换性质。由此即知 M_G 是一个拟阵。

给定一个拟阵 $M = (S,I)$,对于 I 中的一个独立子集 $A \in I$,若 S 有一元素 $x \notin A$,使得将 x 加入 A 后仍保持独立性,即 $A \cup \{x\} \in I$,则称 x 为 A 的一个可扩展元素。

例如,在图拟阵 M_G 中,若 A 是一个独立边集,则边 e 是 A 的一个可扩展元素是指边 e 不在 A 中,且将边 e 加入 A 不会产生循环。

当拟阵 M 中的一个独立子集 A 没有可扩展元素时,称 A 为一个极大独立子集。换句话说,当 A 不被 M 中别的独立子集包含时,A 就是一个极大独立子集。下面的关于极大独立子集的性质是很有用的。

定理 4.1 拟阵 M 中所有极大独立子集具有相同大小。

证明:用反证法。设 A 和 B 是 M 的极大独立子集,且 $|B| > |A|$。由拟阵的交换性质可推出,存在某一元素 $x \in B - A$ 使得 $A \cup \{x\} \in I$。这与 A 是极大独立子集相矛盾。同理,$|A| < |B|$ 也将导致矛盾,故 $|A| = |B|$。

在关于无向图 G 的图拟阵 M_G 中,M_G 的一个极大独立子集是连接图 G 中所有顶点且有 $|V| - 1$ 条边的自由树。这种树就是图 G 的生成树。

若对拟阵 $M = (S,I)$ 中的 S 指定一个权函数 W,使得对于任意 $x \in S$,有 $W(x) > 0$,则称拟阵 M 为带权拟阵。依此权函数,S 的任一子集 A 的权定义为 $W(A) = \sum_{x \in A} W(x)$。

例如,在图拟阵 M_G 中,定义 $W(e)$ 为边 e 的长度,则 $W(A)$ 是边集 A 中所有边的长度之和。

2. 关于带权拟阵的贪心算法

许多可以用贪心算法求解的问题可以表示为求带权拟阵的最大权独立子集问题。即给定一个带权拟阵 $M = (S, I)$，要确定 S 的一个独立子集 $A \in I$ 使得 $W(A)$ 达到最大。这种使 $W(A)$ 最大的独立子集 A 称为拟阵 M 的一个最优子集。由于 S 中任一元素 x 的权 $W(x)$ 是正的，因此，最优子集也一定是极大独立子集。

例如，在最小生成树问题中，我们要找出无向图 $G = (V, E)$ 的一棵生成树，使该树各边长之和达到最小。其中各边的边长由边长函数 W 给出。这个问题可以表示为确定带权拟阵 M_G 的一个最优子集问题。其中，M_G 是图 G 的图拟阵，且权函数 W' 定义为 $W'(e) = W_0 - W(e)$。W_0 是比 G 中最大边长还大的一个正数。M_G 中每一极大独立子集 A 相应于图 G 中一棵生成树，且 $W'(A) = (|V| - 1)W_0 - W(A)$。因此，使权 $W'(A)$ 最大的独立子集 A 必使 $W(A)$ 达到最小。即带权 W' 的 M_G 的最优子集与图 G 的最小生成树之间存在一一对应关系。由此可知，求带权拟阵的最优子集 A 的算法可用于解最小生成树问题。

下面给出一个求带权拟阵最优子集的贪心算法。该算法以具有正权函数 W 的带权拟阵 $M = (S, I)$ 作为输入，经计算后输出 M 的一个最优子集 A。

```
template < class Type >
  Type Greedy(M, W)
  {
      A = ∅;
      将 S 中元素依权 W 值大的优先组织成一个优先队列;
      while (S! = ∅) {
          S.DeleteMax(x);
          if (A ∪ {x} ∈ I) A = A ∪ {x};
          }
      return A
  }
```

算法 Greedy 以贪心选择的方式，按权值从大到小的次序依次考虑 S 中元素 x。当 x 是 A 的一个可扩展元素时，就将 x 加入独立集 A 中，否则舍弃 x。由拟阵的定义，空集是独立的，而且在算法中仅当 $A \cup \{x\}$ 是独立集时才将 x 加入 A，故由归纳法即知 A 总是独立的。因此，算法 Greedy 返回的子集 A 是独立子集。稍后我们将看到 A 是具有最大权的独立子集。因此，A 是一个最优子集。

算法 Greedy 的计算时间可分为两部分来分析。

设 $n = |S|$。将 S 中元素依权值大的优先组织成一个优先队列，并对它进行 n 次 DeleteMax 运算只需要 $O(n\log n)$ 计算时间。若检测 $A \cup \{x\}$ 是否独立需要 $O(f(n))$ 计算时间，则将 S 中所有元素检测一遍需要的计算时间为 $O(nf(n))$。因此，算法 Greedy 的计算时间复杂性为 $O(n\log n + nf(n))$。

下面我们来证明算法 Greedy 的正确性，即它返回的独立子集 A 是 M 的一个最优子集：

引理 4.1 （拟阵的贪心选择性质）

设 $M = (S, I)$ 是具有权函数 W 的一个带权拟阵，且 S 中元素依权值从大到小排列。又设 $x \in S$ 是 S 中第一个使得 $\{x\}$ 是独立子集的元素，则存在 S 的一个最优子集 A 使得 $x \in A$。

证明：若不存在 $x \in S$ 使得 $\{x\}$ 是独立子集，则引理是平凡的。设 B 是一个非空的最优子集。由于 $B \in I$，且 I 具有遗传性，故 B 中所有单个元素 y 组成的子集 $\{y\}$ 均为独立子集。又由于 x 是 S 中的第一个单元素独立子集，故对任意的 $y \in B$ 均有：$W(x) \geq W(y)$。

若 $x \in B$，则只要令 $A = B$，定理得证；若 $x \notin B$，我们将构造包含元素 x 的最优子集 A。一开始，设 $A = \{x\}$，此时，A 是一个独立子集。若 $|B| = |A| = 1$，则定理得证。否则，必有 $|B| > |A|$。反复利用拟阵 M 的交换性质，从 B 中选择一个新元素加入 A 中并保持 A 的独立性，直至 $|B| = |A|$。此时，必有一元素 $y \in B$ 且 $y \notin A$，使得 $A = B - \{y\} \cup \{x\}$。由此知

$$W(A) = W(B) - W(y) + W(x) \geq W(B)$$

另一方面又由于 B 是一个最优子集，故有 $W(B) \geq W(A)$。因此，$W(A) = W(B)$，即 A 也是一个最优子集，且 $x \in A$。

算法 Greedy 在作贪心选择构造最优子集 A 时，首次选入集合 A 中的元素 x 是单元素独立集中具有最大权的元素。此时可能已经舍弃了 S 中部分元素。我们要证明这些被舍弃的元素永远不可能用于构造最优子集。

引理 4.2 设 $M = (S, I)$ 是一个拟阵。若 S 中元素 x 不是空集 $\varnothing$ 的一个可扩展元素，则 x 也不可能是 S 中任一独立子集 A 的一个可扩展元素。

证明：用反证法。设 $x \in S$ 不是 $\varnothing$ 的一个可扩展元素，但它是 S 的独立子集 A 的一个可扩展元素，即 $A \cup \{x\} \in I$。由 I 的遗传性又可推出 $\{x\}$ 是独立的。这与 x 不是空集 $\varnothing$ 的一个可扩展元素相矛盾。

由引理 4.2 即知，算法 Greedy 在初始化独立子集 A 时所舍弃的元素可以永远舍弃。

引理 4.3 （拟阵的最优子结构性质）

设 x 是求带权拟阵 $M = (S, I)$ 的最优子集的贪心算法 Greedy 所选择的 S 中的第一个元素。那么，原问题可简化为求带权拟阵 $M' = (S', I')$ 的最优子集问题，其中

$$S' = \{y \mid y \in S \text{ 且 } \{x, y\} \in I\}$$

$$I' = \{B \mid B \subseteq S - \{x\} \text{ 且 } B \cup \{x\} \in I\}$$

M' 的权函数是 M 的权函数在 S' 上的限制（称 M' 为 M 关于元素 x 的收缩）。

证明：若 A 是 M 的包含元素 x 的最大权独立子集，则 $A' = A - \{x\}$ 是 M' 的一个独立子集。反之，M' 的任一独立子集 A' 产生 M 的一个独立子集 $A = A' \cup \{x\}$。在这两种情形下均有：$W(A) = W(A') + W(x)$。因此 M 的包含元素 x 的最优子集包含 M' 的一个最优子集，反之亦然。

定理 4.2 （带权拟阵贪心算法的正确性）

设 $M = (S, I)$ 是一个具有权函数 W 的带权拟阵，则算法 Greedy 返回 M 的一个最优子集。

证明：由引理 4.1 知，若算法 Greedy 第一次选择加入 A 的元素是 x，则必存在包含元素 x 的一个最优子集。因此，Greedy 的第一次选择是正确的。由引理 4.2 知，选择 x 时 Greedy 所舍弃的元素不可能是任一最优子集中的元素。因此，这些元素可以永远舍弃。最后，由引理 4.3 知，Greedy 选择了元素 x 后，原问题简化为求拟阵 M' 的最优子集问题。由于对于 M' 中任一独立子集 $B \in I'$ 均有 $B \cup \{x\}$ 在 M 中是独立的。因此，Greedy 选择了元素 x 后，其后继步骤可以

看作是对拟阵 $M' = (S', I')$ 进行计算的。由归纳法即知，其后继步骤求出 M' 的一个最优子集，从而算法 Greedy 最终求出的是 M 的一个最优子集。

3. 任务时间表问题

一个单位时间任务是恰好需要一个单位时间来完成的任务。给定一个单位时间任务的有限集 S，关于 S 的一个时间表用于描述 S 中单位时间任务的执行次序。时间表中第 1 个任务从时间 0 开始执行直至时间 1 结束，第 2 个任务从时间 1 开始执行至时间 2 结束，…，第 n 个任务从时间 $n-1$ 开始执行直至时间 n 结束。

具有截止时间和误时惩罚的单位时间任务时间表问题可描述如下。

问题的输入：

(1) n 个单位时间任务的集合 $S = \{1, 2, \cdots, n\}$；

(2) 任务 i 的截止时间 d_i，$1 \leqslant i \leqslant n$，$1 \leqslant d_i \leqslant n$，要求任务 i 在时间 d_i 之前结束。

(3) 任务 i 的误时惩罚 w_i，$1 \leqslant i \leqslant n$，任务 i 未在时间 d_i 之前结束将招致 w_i 的惩罚；若按时完成则无惩罚。

任务时间表问题要求确定 S 的一个时间表(最优时间表)使得总误时惩罚达到最小。这个问题看上去很复杂，然而借助于拟阵，我们可以用带权拟阵的贪心算法有效地求解。对于一个给定的 S 的时间表，其中在截止时间之前完成的任务称为及时任务，在截止时间之后完成的任务称为误时任务。我们可以将 S 的任一时间表调整成为及时优先的形式，即其中所有及时任务先于误时任务，而不影响原时间表中各任务的及时或误时性质。事实上，若时间表中及时任务 x 跟在误时任务 y 之后，则交换 x 和 y 在时间表中的位置不会影响二者的及时或误时性质。通过若干次的这种交换即可将原时间表调整成为及时优先的形式。

类似地，还可将 S 的任一时间表调整成为规范形式，其中及时任务先于误时任务，且及时任务依其截止时间的非减序排列。首先可将时间表调整为及时优先形式，然后再进一步调整及时任务的次序。在时间表中，若有两个及时任务 i 和 j 分别在时间 k 和时间 $k+1$ 完成且 $d_j < d_i$，则交换 i 与 j 在时间表中的位置。由于在交换前任务 j 是及时的，故 $k+1 \leqslant d_j < d_i$。因此在交换位置后 $k+1 < d_i$，即任务 i 仍是及时任务。任务 j 在时间表中位置前移，故交换位置后任务 j 也是及时的。由此可知，这种交换不影响任务 i 和任务 j 的及时性质。经过若干次交换即可将时间表调整成为规范形式。

通过以上的分析可以看出，任务时间表问题可等价于确定最优时间表中及时任务子集 A 的问题。一旦确定了及时任务子集 A，将 A 中各任务依其截止时间的非减序列出，然后再以任意次序列出误时任务，即 $S-A$ 中各任务，由此产生 S 的一个规范的最优时间表。

设 $A \subseteq S$ 是一个任务子集。若有一个时间表使得 A 中所有任务都是及时的，则称 A 为 S 的一个独立任务子集。显然，S 的任一时间表中及时任务构成的集合均为 S 的独立任务子集。记 I 为 S 的所有独立任务子集所构成的集合。

对时间 $t = 1, 2, \cdots, n$，设 $N_t(A)$ 是任务子集 A 中所有截止时间是 t 或更早的任务数。我们来考虑如何判断任务子集 A 的独立性。

引理 4.4 对于 S 的任一任务子集 A，下面的各命题是等价的。

(1) 任务子集 A 是独立子集。

(2) 对于 $t = 1, 2, \cdots, n$，都有 $N_t(A) \leqslant t$。

(3) 若 A 中任务依其截止时间非减序排列，则 A 中所有任务都是及时的。

证明:(1)⇒(2):若任务集 A 是独立的,且存在某个 t 使得 $N_t(A) > t$,则 A 中有多于 t 个任务要在时间 t 之前完成,显然这是办不到的。故 A 中必有误时任务。这与 A 是独立任务子集矛盾。因此,对所有 $t = 1,2,\cdots,n$ 有 $N_t(A) \leqslant t$。

(2)⇒(3):若 A 中任务依其截止时间的非减序排列,则(2)中不等式意味着排序后 A 中第 i 个任务的截止时间在时间 i 之后。故排序后 A 中所有任务都是及时的。

(3)⇒(1):显而易见。

引理4.4中的性质(2)可用于有效地判断一个给定的任务子集的独立性。

任务时间表问题要求使总误时惩罚达到最小,而这等价于使任务时间表中的及时任务的惩罚值之和达到最大。下面的定理使我们能用带权拟阵上的贪心算法求出总惩罚最大的独立任务子集 A。

引理4.5 设 S 是带有截止时间的单位时间任务集,I 是 S 的所有独立任务子集构成的集合。则有序对 (S,I) 是一个拟阵。

证明:独立任务集的子集显然也是独立子集。故 I 满足遗传性质。下面证明 (S,I) 满足交换性质。

设 A 和 B 为两个独立任务子集且 $|B| > |A|$。设 $k = \max\limits_{1 \leqslant t \leqslant n}\{t \mid N_t(B) \leqslant N_t(A)\}$。由于 $N_n(B) = |B|$,$N_n(A) = |A|$,而 $|B| > |A|$,即 $N_n(B) > N_n(A)$。因此必有 $k < n$,且对于满足 $k+1 \leqslant j \leqslant n$ 的 j 有 $N_j(B) > N_j(A)$。取 $x \in B - A$ 且 x 的截止时间为 $k+1$。令 $A' = A \cup \{x\}$。我们来证明 A' 是独立的。

事实上,由于 A 是独立的,故对 $1 \leqslant t \leqslant k$ 有 $N_t(A') = N_t(A) \leqslant t$。又由于 B 是独立的,故对 $k < t \leqslant n$ 有 $N_t(A') = N_t(A) + 1 \leqslant N_t(B) \leqslant t$。由引理4.4即知 A' 是独立的。综上所述即知,(S,I) 是一个拟阵。

由定理4.2可知用带权拟阵的贪心算法可以求得最大权(惩罚)独立任务子集 A,以 A 作为最优时间表中的及时任务子集,容易构造一个最优时间表。

用于求解任务时间表问题的贪心算法的计算时间复杂性是 $O(n\log n + nf(n))$。其中,$f(n)$ 是用于检测任务子集 A 的独立性所需的计算时间。用引理4.4中性质(2)容易设计一个 $O(n)$ 时间算法来检测任务子集的独立性。因此,整个算法的计算时间为 $O(n^2)$。具体算法GreedyJob可描述如下。其中,d[i],$1 \leqslant i \leqslant n$,是 n 个单位时间任务的截止时间,且 n 个单位时间任务已依其误时惩罚的非增序排列。J[i] 是最优解中的第 i 个任务。

```
int GreedyJob(int n,int d[],int J[])
  {
      d[0] = 0;J[0] = 0;
      int k = 1;
      J[1] = 1;
      for (int i = 2;i < = n;i ++) {
          int r = k;
          while ((d[J[r]] > d[i])&&(d[J[r]]! = r))
            r = r - 1;
          if ((d[J[r]] < = d[i])&&(d[i] > r)) {
            for (int m = k;m > r;m --)
```

```
            J[m + 1] = J[m];
          J[r + 1] = i;
          k = k + 1;


      }
    return k;
}
```

例如,给定单位时间任务集 S 及各任务的截止时间和误时惩罚如下:

i	1	2	3	4	5	6	7
d[i]	4	2	4	3	1	4	6
w[i]	70	60	50	40	30	20	10

算法 GreedyJob 先选择任务 1,2,3,4,然后舍弃任务 5,6,最后再选择任务 7。算法求得的最优时间表为{2,4,1,3,7,5,6}。其总误时惩罚为 $w[5]+w[6]=50$,达到最小。

用抽象数据类型并查集 UnionFind 可对上述算法作进一步改进。采用相同的贪心选择策略来选择任务。为了给后继任务留下尽可能大的选择空间,在选择了任务 i 时,将[0,1],[1,2],…,$[d_i-1,d_i]$ 中最右端的空闲时间区间分配给任务 i。任何一个最优时间表最多只能安排 $b=\min\{n,\max\limits_{1\leqslant i\leqslant n}\{d_i\}\}$ 个及时任务。为方便起见,直接用 i 来表示时间区间$[i-1,i]$,以 0 表示左端空闲区间$[-1,0]$。设 n_i 表示小于或等于 i 的最右端空闲区间,则 $n_i\leqslant i$。我们将时间区间划分为一些等价类。时间区间 i 和 j 属于同一等价类当且仅当 $n_i=n_j$。该等价类就以 n_i 命名。初始时有 $0,1,\cdots,b$ 共 $b+1$ 个等价类。要安排截止时间为 d 的任务,先用 Find 找到含有时间区间 $\min\{n,d\}$ 的等价类 k,F_k 就表示可以安排当前任务的最右端的那个空闲时间区间。安排完之后,等价类 k 就应与含有时间区间 F_k-1 的等价类用 Union 合并。

改进后的算法 FasterJob 描述如下:

```
int FasterJob(int n,int d[],int J[])
{
   int * F = new int [n + 1];
   for (int i = 0;i < = n;i ++) F[i] = i;
   UnionFind U(n);
   int k = 0;
   for (int i = 1;i < = n;i ++) {
     int m = (n < d[i])?U.Find(n):U.Find(d[i]);
     if (F[m] > 0) {
       k = k + 1;
       J[k] = i;
       int t = U.Find(F[m] - 1);
       U.Union(t,m);
       F[m] = F[t];
       }
```

```
        }
        return k;
    }
...
```

算法 FasterJob 用到的 Find 和 Union 运算的次数都不超过 n 次。因此,如果不计预处理的时间,算法 FasterJob 所需的计算时间为 $O(n\log^* n)$。

习题 4

4-1　假设我们要在足够多的会场里安排一批活动,并希望使用尽可能少的会场。设计一个有效的贪心算法来进行安排(这个问题实际上是著名的图着色问题。若将每一个活动作为图的一个顶点,不相容的活动间用边相连。则使相邻顶点着有不同颜色的最小着色数,就相应于我们要找的最小会场数)。

4-2　在活动安排问题中,还可以有其他的贪心选择方案,但并不能保证产生最优解。给出一个例子,说明若选择具有最短时段的相容活动作为贪心选择,得不到最优解。若选择覆盖未选择活动最少的相容活动作为贪心选择,也得不到最优解。

4-3　证明背包问题具有贪心选择性质。

4-4　若在 0-1 背包问题中,各物品依重量递增排列时,其价值恰好依递减序排列。对这个特殊的0-1 背包问题,设计一个有效算法找出最优解,并说明算法的正确性。

4-5　给定 k 个排好序的有序序列 $s_1,s_2,\cdots,s_k$,现要用2路合并算法将这 k 个序列合并成一个有序序列。假设所采用的2路合并算法合并两个长度分别为 m 和n 的序列需要$m+n-1$ 次比较。试设计一个算法确定合并这个序列的最优合并顺序,使得所需的总比较次数最少。

4-6　设有 n 个程序$\{1,2,\cdots,n\}$要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 l_i,$1\le i\le n$。如果将这 n 个程序按 $i_1,i_2,\cdots,i_n$ 的次序存放,则读取 i_r 程序所需的时间与 $\sum_{k=1}^{r} l_{i_k}$ 成正比。这 n 个程序的平均读取时间为$\left(\sum_{r=1}^{n} t_r\right)/n$。

磁带最优存储问题要求确定这 n 个程序在磁带上的一个存储次序,使平均读取时间达到最小。试设计一个解此问题的算法,并分析算法的正确性和计算复杂性。

4-7　设有 n 个文件 $f_1,f_2,\cdots,f_n$ 要求存放在一个磁盘上,每个文件占磁盘上 1 个磁道。这 n 个文件的检索概率分别是 $p_1,p_2,\cdots,p_n$,且 $\sum_{i=1}^{n} p_i=1$。磁头从当前磁道移到被检信息磁道所需的时间可用这两个磁道之间的径向距离来度量。如果文件 f_i 存放在第 i 道上,$1\le i\le n$ 则检索这 n 个文件的期望时间是 $\sum_{1\le i<j\le n} p_i p_j d(i,j)$。其中,$d(i,j)$ 是第 i 道与第j 道之间的径向距离。

磁盘文件的最优存储问题要求确定这 n 个文件在磁盘上的存储位置,使期望检索时间达到最小。试设计一个解此问题的算法,并分析算法的正确性与计算复杂性。

4-8　设 $p_1,p_2,\cdots,p_n$ 是要存放在一条长度为 L 的磁带上的 n 个程序。每个程序 p_i 所需的带长为 a_i。如果$\sum a_i\le L$,则所有的程序都能放在一条磁带上。如果假定$\sum a_i>L$,要求找出这些程序的最大子集 Q,使得 Q 中的程序能存放在一条磁带上。其中最大子集 Q 的定义是

Q 中包括的程序个数最多。

(1) 设 p_i 的顺序满足 $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$，写一个求最大子集 Q 的算法。要求输出结果是数组 s。如果程序 p_i 在 Q 中，则 s[i] = 1；否则 s[i] = 0。

(2) 证明你设计的算法能保证找到 $p_1, p_2, \cdots, p_n$ 的使 $\sum_{p_i \in Q} a_i \leqslant L$ 的最大子集 Q。

(3) 设 Q 是按上面的策略得到的最大子集，磁带的利用率 $\left(\sum_{p_i \in Q} a_i\right)/L$ 有多大?

(4) 假定现在的目标是要求磁带的利用率最大，而程序 p_i 的顺序满足 $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_n$。要求按 $p_1, p_2, \cdots, p_i \cdots$ 的顺序来考虑选取 p_i 到 Q 中，只要磁带上的剩余空间足够容纳 p_i，就应当把 p_i 选入 Q 中。按以上策略写一个算法并分析其时间和空间复杂性。

(5) 证明按(4)的策略所得到的子集未必能使带的利用率达到最大。利用率能小到什么程度?试证明你设计的算法的界。

4-9　问题的条件如上题，现在的目标是：① 使子集 Q 达到最大；② 在保证 Q 最大的前提下使带的利用率达到最大。应当采用什么策略?写出一个完整的算法并证明其正确性。

4-10　假定要把长为 $l_1, l_2, \cdots, l_n$ 的 n 个程序放在磁带 T_1 和 T_2 上，并且希望按照使最大检索时间取最小值的方式存放，即如果存放在 T_1 和 T_2 上的程序集合分别是 A 和 B，则希望所选择的 A 和 B 使得 $\max\left\{\sum_{i \in A} l_i, \sum_{i \in B} l_i\right\}$ 取最小值。一种得到 A 和 B 的贪心算法如下：开始将 A 和 B 都初始化为空，然后一次考虑一个程序，如果 $\sum_{i \in A} l_i = \min\left\{\sum_{i \in A} l_i, \sum_{i \in B} l_i\right\}$，则将当前正在考虑的那个程序分配给 A，否则分配给 B。证明无论是按 $l_1 \leqslant l_2 \leqslant \cdots \leqslant l_n$ 或是按 $l_1 \geqslant l_2 \geqslant \cdots \geqslant l_n$ 的次序来考虑程序，这种方法都不能产生最优解。应当采用什么策略?写出一个完整的算法并证明其正确性。

4-11　设有 n 个顾客同时等待一项服务。顾客 i 需要的服务时间为 t_i，$1 \leqslant i \leqslant n$，应如何安排 n 个顾客的服务次序才能使总的等待时间达到最小?总的等待时间是每个顾客等待服务时间的总和。

4-12　在上题中，如果有 s 处提供同一服务，应如何安排 n 个顾客的服务次序?

4-13　将最优装载问题的贪心算法推广到 2 艘船的情形，贪心算法仍能产生最优解吗?

4-14　设 T 是一棵带权树，树的每一条边带一个正权。又设 S 是 T 的顶点集，T/S 是从树 T 中将 S 中顶点删去后得到的森林。如果 T/S 中所有树的从根到叶的路长都不超过 d，则称 T/S 是一个 d 森林。

(1) 设计一个算法求 T 的一个最小顶点集 S，使 T/S 是一个 d 森林(提示：从叶向根移动)。

(2) 分析算法的正确性和计算复杂性。

(3) 设 T 中有 n 个顶点，则算法的计算时间复杂性应为 $O(n)$。

4-15　将任务安排问题的贪心算法推广到完成任务 i 需要 t_i 时间的情形，$1 \leqslant i \leqslant n$。

4-16　一辆汽车加满油后可行驶 n 千米。旅途中有若干个加油站。若要使沿途加油次数最少，设计一个有效算法，指出应在哪些加油站停靠加油，并证明你的算法能产生一个最优解。

4-17　设 $x_1, x_2, \cdots, x_n$ 是实直线上的 n 个点。若要用单位长度的闭区间去覆盖这 n 个点，至少需要多少个这样的单位闭区间?设计一个有效算法解此问题，并证明算法的正确性。

4-18　字符 a ~ h 出现的频率恰好是前 8 个 Fibonacci 数，它们的哈夫曼编码是什么?将结

果推广到 n 个字符的频率恰好是前 n 个 Fibonacci 数的情形。

4-19　设 $C=\{0,1,\cdots,n-1\}$ 是 n 个字符的集合。证明关于 C 的任何最优前缀码可以表示为长度为 $2n-1+n\lceil\log n\rceil$ 位的编码序列。(提示：用 $2n-1$ 位来描述树结构。)

4-20　将哈夫曼算法推广到三元码的情形(即用 0,1 和 2 进行编码)，并证明算法可产生最优三元码。

4-21　说明如何用引理 4.4 的性质(2)，在 $O(|A|)$ 时间里确定给定的任务集 A 是否独立。

4-22　给定一个 $n\times n$ 实值矩阵 $\boldsymbol{T}$，证明 (S,I) 是一个拟阵。其中，S 是 $\boldsymbol{T}$ 的列向量的集合，$A\in I$ 当且仅当 A 中的列是线性独立的。

4-23　说明如何变换带权拟阵的权函数，使得求最小权最大独立子集问题变换为等价的标准带权拟阵问题，并证明变换的正确性。

4-24　考虑下面的用最少硬币个数找出 n 分钱的问题。

(1) 当使用 2 角 5 分，1 角，5 分和 1 分 4 种硬币面值时，设计一个找 n 分钱的贪心算法，并证明算法能产生一个最优解。

(2) 假设可使用的硬币面值是 $c^0,c^1,\cdots,c^k$，其中，c 是一正整数且 $c>1$、$k\geqslant 1$。证明在这种情况下，贪心算法总能产生最优解。

(3) 给出一个贪心算法不能产生最优解的硬币面值集合。

4-25　给定一个 n 位正整数 a，去掉其中任意 $k\leqslant n$ 个数字后，剩下的数字按原次序排列组成一个新的正整数。对于给定的 n 位正整数 a 和正整数 k，设计一个算法找出剩下数字组成的新数最小的删数方案。

4-26　在黑板上写了 n 个正数组成的一个数列，进行如下操作：每一次擦去其中两个数设为 a 和 b，然后在数列中加入一个数 $a*b+1$，如此下去直至黑板上只剩下一个数。在所有按这种操作方式最后得到的数中，最大的数记为 max，最小的数记为 min，则该数列的极差 M 定义为 $M=\max-\min$。对于给定的数列，设计一个有效算法计算出其极差 M，并说明算法的正确性。

4-27　假设具有 n 个顶点的连通带权图中所有边的权值均为从 1 到 n 之间的整数，那么你能使 Kruskal 算法作何改进，时间复杂性能改进到何种程度？若对某常量 N，所有边的权值均为从 1 到 N 之间的整数，在这种情况下又如何？在上述两种情况下，对 Prim 算法能作何改进？

4-28　试设计一个构造图 G 的生成树的算法，使得构造出的生成树的边的最大权值达到最小。

4-29　试举例说明如果允许带权有向图中某些边的权为负实数，则 Dijkstra 算法不能正确地求出从源到所有其他顶点的最短路径长度。

4-30　设 G 是一个具有 n 个顶点和 e 条边的带权有向图，各边的权值为 0 到 $N-1$ 之间的整数，N 为一非负整数。修改 Dijkstra 算法使其能在 $O(Nn+e)$ 时间内计算出从源到所有其他顶点之间的最短路径长度。

4-31　一个 d 维箱 $(x_1,x_2,\cdots,x_d)$ 嵌入另一个 d 维箱 $(y_1,y_2,\cdots,y_d)$ 是指，存在 $1,2,\cdots,d$ 的一个排列 π，使得 $x_{\pi(1)}<y_1,x_{\pi(2)}<y_2,\cdots,x_{\pi(d)}<y_d$。

(1) 证明上述箱嵌套关系具有传递性。

(2) 试设计一个有效算法,用于确定一个 d 维箱是否可嵌入另一个 d 维箱。

(3) 给定由 n 个 d 维箱组成的集合 $\{B_1, B_2, \cdots, B_n\}$,试设计一个有效算法找出这 n 个 d 维箱中的一个最长嵌套箱序列,并用 n 和 d 来描述算法的计算时间复杂性。

4-32 套汇是指利用货币汇兑率的差异将一个单位的某种货币转换为大于一个单位的同种货币。例如,假定 1 美元可以买 0.7 英镑,1 英镑可以买 9.5 法郎,且 1 法郎可以买到 0.16 美元。通过货币兑换,一个商人可以从 1 美元开始买入,得到 $0.7 \times 9.5 \times 0.16 = 1.064$ 美元,从而获得 6.4%的利润。

假设已知 n 种货币 $c_1, c_2, \cdots, c_n$ 和有关兑换率的 $n \times n$ 表 R。其中,$R[i, j]$ 是一个单位货币 c_i 可以买到的货币 c_j 的单位数。

(1) 试设计一个有效算法,用以确定是否存在一货币序列 $c_{i1}, c_{i2}, \cdots, c_{ik}$ 使得

$$R[i1, i2]R[i2, i3]\cdots R[ik, i1] > 1$$

并分析算法的计算时间。

(2) 试设计一个算法打印出满足(1) 中条件的所有序列,并分析算法的计算时间。

第5章　回　溯　法

学习要点

· 理解回溯法的深度优先搜索策略
· 掌握用回溯法解题的算法框架：
(1) 递归回溯最优子结构性质
(2) 迭代回溯贪心选择性质
(3) 子集树算法框架
(4) 排列树算法框架
· 通过下面的应用范例学习回溯法的设计策略：
(1) 装载问题
(2) 批处理作业调度
(3) 符号三角形问题
(4) n后问题
(5) 0－1背包问题
(6) 最大团问题
(7) 图的m着色问题
(8) 旅行售货员问题
(9) 圆排列问题
(10) 电路板排列问题
(11) 连续邮资问题

回溯法有"通用的解题法"之称。用它可以系统地搜索一个问题的所有解或任一解。回溯法是一个既带有系统性又带有跳跃性的搜索算法。它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树。算法搜索至解空间树的任一结点时，总是先判断该结点是否肯定不包含问题的解。如果肯定不包含，则跳过对以该结点为根的子树的系统搜索，逐层向其祖先结点回溯。否则，进入该子树，继续按深度优先的策略进行搜索。回溯法在用来求问题的所有解时，要回溯到根，且根结点的所有子树都已被搜索遍才结束。而回溯法在用来求问题的任一解时，只要搜索到问题的一个解就可结束。这种以深度优先的方式系统地搜索问题的解的算法称为回溯法，它适用于解一些组合数较大的问题。

5.1　回溯法的算法框架

1. 问题的解空间

应用回溯法解问题时，首先应明确定义问题的解空间。问题的解空间应至少包含问题的一个(最优)解。例如，对于有 n 种可选择物品的0-1背包问题，其解空间由长度为 n 的0-1向量

组成。该解空间包含了对变量的所有可能的 0-1 赋值。当 $n = 3$ 时,其解空间是

$$\{(0,0,0),(0,1,0),(0,0,1),(1,0,0),(0,1,1),(1,0,1),(1,1,0),(1,1,1)\}$$

定义了问题的解空间后,还应将解空间很好地组织起来,使得用回溯法能方便地搜索整个解空间。通常我们将解空间组织成树或图的形式。

例如,对于 $n = 3$ 时的 0-1 背包问题,其解空间用一棵完全二叉树表示,如图 5-1 所示。

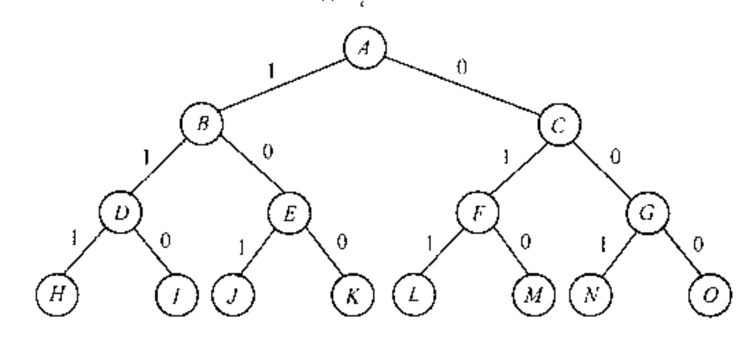


图 5-1　0-1 背包问题的解空间树

解空间树的第 i 层到第 $i+1$ 层边上的标号给出了变量的值。从树根到叶的任一路径表示解空间中的一个元素。例如,从根结点到结点 H 的路径相应于解空间中的元素(1,1,1)。

2. 回溯法的基本思想

确定了解空间的组织结构后,回溯法就从开始结点(根结点)出发,以深度优先的方式搜索整个解空间。这个开始结点就成为一个活结点,同时也成为当前的扩展结点。在当前的扩展结点处,搜索向纵深方向移至一个新结点。这个新结点就成为一个新的活结点,并成为当前扩展结点。如果在当前的扩展结点处不能再向纵深方向移动,则当前的扩展结点就成为死结点。换句话说,这个结点不再是一个活结点。此时,应往回移动(回溯)至最近的一个活结点处,并使这个活结点成为当前的扩展结点。回溯法即以这种工作方式递归地在解空间中搜索,直至找到所要求的解或解空间中已无活结点时为止。

例如,对于 $n = 3$ 时的 0-1 背包问题,考虑下面的具体实例:$w = [16,15,15]$,$p = [45,25,25]$,$c = 30$。我们从图 5-1 的根结点开始搜索其解空间。开始时根结点是惟一的活结点,也是当前的扩展结点。在这个扩展结点处,我们可以沿纵深方向移至结点 B 或结点 C。假设我们选择先移至结点 B。此时,结点 A 和结点 B 是活结点,结点 B 成为当前扩展结点。由于选取了 w_1,故在结点 B 处剩余背包容量是 $r = 14$,获取的价值为 45。从结点 B 处,我们可以移至结点 D 或 E。由于移至结点 D 至少需要 $w_2 = 15$ 的背包容量,而我们现在仅有的背包容量是 $r = 14$,故移至结点 D 导致一个不可行解。而搜索至结点 E 不需要背包容量,因而是可行的。从而我们选择移至结点 E。此时,E 成为新的扩展结点,结点 A、B 和 E 是活结点。在结点 E 处,$r = 14$,获取的价值为 45。从结点 E 处,可以向纵深移至结点 J 或 K。移至结点 J 导致一个不可行解,而移向结点 K 是可行的,于是移向结点 K,它成为一个新的扩展结点。由于结点 K 是一个叶结点,故我们得到一个可行解。这个解相应的价值为 45。x_i 的取值由根结点到叶结点 K 的路径所惟一确定,即 $x = (1,0,0)$。由于在结点 K 处已不能再向纵深扩展,所以结点 K 成为死结点。我们返回到结点 E 处。此时在结点 E 处也没有可扩展的结点,它也成为死结点。

接下来我们又返回到结点 B 处。结点 B 同样也成为死结点,从而结点 A 再次成为当前扩展结点。结点 A 还可继续扩展,从而到达结点 C。此时,$r = 30$,获取的价值为 0。从结点 C 我们

可移向结点 F 或 G。假设我们移至结点 F，它成为新的扩展结点。结点 A，C 和 F 是活结点。在结点 F 处，$r = 15$，获取的价值为 25。从结点 F，我们向纵深移至结点 L 处，此时，$r = 0$，获取的价值为 50。由于 L 是一个叶结点，而且是迄今为止找到的获取价值最高的可行解，因此记录这个可行解。结点 L 不可扩展，我们又返回到结点 F 处。按此方式继续搜索，可搜索遍整个解空间。搜索结束后找到的最好解是相应 0-1 背包问题的最优解。

我们再看一个用回溯法解旅行售货员问题的例子。

旅行售货员问题的提法是：某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。他要选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使总的路程(或总旅费)最小。

问题刚提出时，不少人都认为这个问题很简单。后来，人们在实践中才逐步认识到，这个问题只是叙述简单，易于为人们所理解，而其计算复杂性却是问题的输入规模的指数函数。属于相当难解的问题之一。事实上，它是一个 NP 完全问题。这个问题可以用图论的语言来进行形式描述。

设 $G = (V, E)$ 是一个带权图。图中各边的费用(权)为一正数。图中的一条周游路线是包括 V 中的每个顶点在内的一条回路。一条周游路线的费用是这条路线上所有边的费用之和。所谓旅行售货员问题就是要在图 G 中找出一条有最小费用的周游路线。

给定一个有 n 个顶点的带权图 G，旅行售货员问题要找出图 G 的费用(权)最小的周游路线。图 5-2 是一个 4 顶点无向带权图。顶点序列 1,2,4,3,1；1,3,2,4,1 和 1,4,3,2,1 是该图中 3 条不同的周游路线。

该问题的解空间可以组织成一棵树，从树的根结点到任一叶结点的路径定义了图 G 的一条周游路线。图 5-3 是当 $n = 4$ 时这种树结构的示例。其中从根结点 A 到叶结点 L 的路径上边的标号组成一条周游路线 1,2,3,4,1。而从根结点 A 到叶结点 O 的路径则表示周游路线 1,3,4,2,1。图 G 的每一条周游路线都恰好对应于解空间树中一条从根结点到叶结点的路径。因此，解空间树中叶结点个数为 $(n-1)!$。

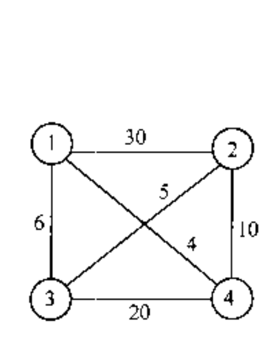


图 5-2　4 顶点带权图

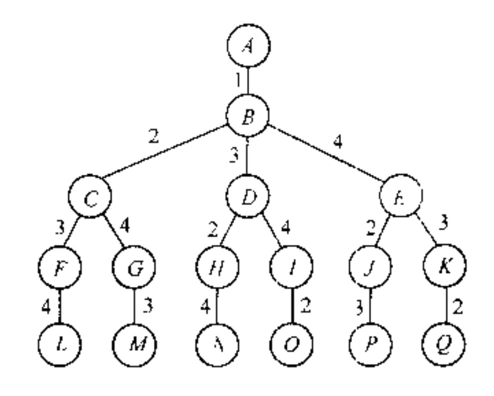


图 5-3　旅行售货员问题的解空间树

对于图 5-2 中的图 G，用回溯法找最小费用周游路线时，从解空间树的根结点 A 出发，搜索至 B, C, F, L。在叶结点 L 处记录找到的周游路线 1,2,3,4,1，该周游路线的费用为 59。从叶结点 L 返回至最近活结点 F 处。由于 F 处已没有可扩展结点，算法又返回到结点 C 处。结点 C 成为新扩展结点，由新扩展结点，算法再移至结点 G 后又移至结点 M，得到周游路线 1,2,4,3,1，其费用为 66。这个费用不比已有周游路线 1,2,3,4,1 的费用更小。因此，舍弃该结点。算法又依次返回至结点 G, C, B。从结点 B，算法继续搜索至结点 D, H, N。在叶结点 N 处，相应的

周游路线 1,3,2,4,1 的费用为 25。它成为迄今为止找到的最好的一条周游路线。从结点 N 算法返回至结点 H, D,然后再从结点 D 开始继续向纵深搜索至结点 O。依此方式算法继续搜索遍整个解空间,最终得到 1,3,2,4,1 是一条最小费用周游路线。

在用回溯法搜索解空间树时,通常采用两种策略来避免无效搜索,提高回溯法的搜索效率。其一是用约束函数在扩展结点处剪去不满足约束的子树;其二是用限界函数剪去不能得到最优解的子树。这两类函数统称为剪枝函数。

例如,在解 0-1 背包问题的回溯法中,用剪枝函数剪去导致不可行解的子树。在解旅行售货员问题的回溯法中,如果从根结点到当前扩展结点处的部分周游路线的费用已超过当前找到的最好的周游路线费用,则可以断定以该结点为根的子树中不含最优解,因此可将该子树剪去。

综上所述,运用回溯法解题通常包含以下三个步骤:

(1) 针对所给问题,定义问题的解空间;

(2) 确定易于搜索的解空间结构;

(3) 以深度优先的方式搜索解空间,并且在搜索过程中用剪枝函数避免无效搜索。

3. 递归回溯

由于回溯法是对解空间的深度优先搜索,因此在一般情况下可用递归函数来实现回溯法如下:

```
void Backtrack(int t)
{
   if (t > n) Output(x);
      else
         for (int i = f(n,t);i < = g(n,t);i ++) {
            x[t] = h(i);
            if (Constraint(t) & & Bound(t)) Backtrack(t + 1);
         }
}
```

其中,形式参数 t 表示递归深度,即当前扩展结点在解空间树中的深度。n 用来控制递归深度,即解空间树的高度。当 $t > n$ 时,算法已搜索到一个叶结点。此时,由函数 Output(x) 对得到的可行解 x 进行记录或输出处理。算法 Backtrack 的 for 循环中 f(n, t) 和 g(n, t) 分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。h(i) 表示在当前扩展结点处 x[t] 的第 i 个可选值。函数 Constraint(t) 和 Bound(t) 表示在当前扩展结点处的约束函数和限界函数。函数 Constraint(t) 返回的值为 true 则表示在当前扩展结点处 x[1:t] 的取值满足问题的约束条件,否则不满足问题的约束条件,可剪去相应的子树。函数 Bound(t) 返回的值为 true 则表示在当前扩展结点处 x[1:t] 的取值尚未使目标函数越界,还需由 Backtrack(t + 1) 对其相应的子树作进一步搜索。否则,当前扩展结点处 x[1:t] 的取值已使目标函数越界,可剪去相应的子树。执行了算法的 for 循环后,已搜索遍当前扩展结点的所有未搜索过的子树。Backtrack(t) 执行完毕,返回 t - 1 层继续执行,对还没有测试过的 x[t - 1] 的值继续搜索。当 t = 1 时,若已

测试完 x[1] 的所有可选值,外层调用就全部结束。显然,这一搜索过程是按深度优先的方式进行的。调用一次 Backtrack(1) 即可完成整个回溯搜索过程。

4. 迭代回溯

如果采用树的非递归深度优先遍历算法,也可将回溯法表示为一个非递归的迭代过程如下:

```
void IterativeBacktrack(void)
{
  int t = 1;
  while (t > 0) {
    if (f(n,t) <= g(n,t))
      for (int i = f(n,t);i <= g(n,t);i++) {
        x[t] = h(i);
        if (Constraint(t) && Bound(t)) {
          if (Solution(t)) Output(x);
          else t++;}
        }
      else t--;
    }
}
```

在上述迭代回溯算法的描述中,用函数 Solution(t) 判断在当前扩展结点处是否已得到问题的一个可行解。函数 Solution(t) 返回的值为 true 则表示在当前扩展结点处 x[1:t] 是问题的一个可行解。此时,由函数 Output(x) 对得到的可行解 x 进行记录或输出处理。函数 Solution(t) 返回的值为 false 则表示在当前扩展结点处 x[1:t] 只是问题的一个部分解,还需向纵深方向继续搜索。算法中的函数 f(n,t) 和 g(n,t) 分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。h(i) 表示在当前扩展结点处 x[t] 的第 i 个可选值。函数 Constraint(t) 和 Bound(t) 表示在当前扩展结点处的约束函数和限界函数。函数 Constraint(t) 返回的值为 true 则表示在当前扩展结点处 x[1:t] 的取值满足问题的约束条件,否则不满足问题的约束条件,可剪去相应的子树。函数 Bound(t) 返回的值为 true 则表示在当前扩展结点处 x[1:t] 的取值尚未使目标函数越界,还需对其相应的子树作进一步搜索。否则,当前扩展结点处 x[1:t] 的取值已使目标函数越界,可剪去相应的子树。算法的 while 循环结束后,完成整个回溯搜索过程。

用回溯法解题的一个显著特征是问题的解空间是在搜索过程中动态产生的。在任何时刻,算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为 $h(n)$,则回溯法所需的计算空间通常为 $O(h(n))$。而显式地存储整个解空间则需要 $O(2^{h(n)})$ 或 $O(h(n)!)$ 的空间。

5. 子集树与排列树

图 5-1 和图 5-3 中的两棵解空间树是用回溯法解题时常遇到的两类典型的解空间树。

当所给的问题是从 n 个元素的集合 S 中找出满足某种性质的子集时，相应的解空间树称为子集树。例如，n 个物品的 0-1 背包问题所相应的解空间树就是一棵子集树。这类子集树通常有 2^n 个叶结点，其结点总个数为 $2^{n+1}-1$。遍历子集树的任何算法均需 $\Omega(2^n)$ 的计算时间。

当所给的问题是确定 n 个元素满足某种性质的排列时，相应的解空间树称为排列树。排列树通常有 $n!$ 个叶结点。因此遍历排列树需要 $\Omega(n!)$ 的计算时间。图 5-3 中旅行售货员问题的解空间树就是一棵排列树。

用回溯法搜索子集树的一般算法可描述如下：

```
void Backtrack(int t)
{
  if (t > n) Output(x);
    else
      for (int i = 0;i <= 1;i++) {
            x[t] = i;
            if (Constraint(t)&&Bound(t)) Backtrack(t + 1);
      }
}
```

用回溯法搜索排列树的算法框架可描述如下：

```
void Backtrack(int t)
{
  if (t > n) Output(x);
    else
      for (int i = t;i <= n;i++) {
        Swap(x[t], x[i]);
        if (Constraint(t)&&Bound(t)) Backtrack(t + 1);
        Swap(x[t], x[i]);
      }
}
```

在调用 Backtrack(1) 执行回溯搜索之前，先将变量数组 x 初始化为单位排列 $(1,2,\cdots,n)$。

5.2 装载问题

在第 4 章中我们讨论了最优装载问题的贪心算法。本节中讨论的装载问题是最优装载问题的一个变形。

1. 问题描述

有一批共 n 个集装箱要装上 2 艘载重量分别为 c_1 和 c_2 的轮船，其中集装箱 i 的重量为 w_i，

且 $\sum_{i=1}^{n} w_i \leq c_1 + c_2$。

装载问题要求确定,是否有一个合理的装载方案可将这 n 个集装箱装上这 2 艘轮船。如果有,找出一种装载方案。

例如,当 $n = 3, c_1 = c_2 = 50$,且 $w = [10,40,40]$ 时,则可以将集装箱 1 和 2 装到第一艘轮船上,而将集装箱 3 装到第二艘轮船上;如果 $w = [20,40,40]$,则无法将这 3 个集装箱都装上轮船。

当 $\sum_{i=1}^{n} w_i = c_1 + c_2$ 时,装载问题就等价于子集和问题。当 $c_1 = c_2$ 且 $\sum_{i=1}^{n} w_i = 2c_1$,则装载问题等价于划分问题。

即使限制 $w_i, i = 1,\cdots,n$ 为整数,c_1 和 c_2 也是整数。子集和问题与划分问题都是 NP 难的。由此可知装载问题也是 NP 难的。

容易证明,如果一个给定的装载问题有解,则采用下面的策略可以得到一个最优装载方案。

(1) 首先将第一艘轮船尽可能装满;

(2) 然后将剩余的集装箱装到第二艘轮船上。

将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集,使该子集中集装箱重量之和最接近 c_1。由此可知,装载问题等价于以下特殊的 0-1 背包问题:

$$\max \sum_{i=1}^{n} w_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \leq c_1$$

$$x_i \in \{0,1\}, 1 \leq i \leq n$$

我们可以用第 3 章中讨论过的动态规划算法解这个特殊的 0-1 背包问题。所需的计算时间是 $O(\min\{c_1, 2^n\})$。下面我们用回溯法设计一个解装载问题的 $O(2^n)$ 计算时间算法。在某些情况下该算法优于动态规划算法。

2. 算法设计

用回溯法解装载问题时,用子集树表示其解空间显然是最合适的。用可行性约束函数可剪去不满足约束条件 $\sum_{i=1}^{n} w_i x_i \leq c_1$ 的子树。在子集树的第 $j+1$ 层的结点 Z 处,用 cw 记当前的装载重量,即 $cw = \sum_{i=1}^{j} w_i x_i$,则当 $cw > c_1$ 时,以结点 Z 为根的子树中所有结点都不满足约束条件,因而该子树中的解均为不可行解,故可将该子树剪去。

在下面所给出的解装载问题的回溯法描述中,函数 MaxLoading 返回不超过 c 的最大子集和,但并未给出达到这个最大子集和的相应子集。稍后将使其进一步完善。

函数 MaxLoading 中调用递归函数 Backtrack(1) 实现对整个解空间的回溯搜索。Backtrack(i) 搜索子集树中第 i 层子树。函数 Backtrack 是类 Loading 的成员。类 Loading 的其他成员记录子集树中结点信息,以减少传给函数 Backtrack 的参数。cw 记录当前结点所相应的装载重量,bestw 记录当前已找到的最大装载重量。函数 MaxLoading 负责类 Loading 的私有变量的初始化。

在函数 Backtrack 中，当 $i > n$ 时，表示算法已搜索至一个叶结点，其相应的装载重量为 cw。如果 cw > bestw，则表示当前解优于迄今所找到的最优解，此时应更新 bestw。

当 $i \leqslant n$ 时，当前扩展结点 Z 是子集树中的一个内部结点。该结点有 $x[i]=1$ 和 $x[i]=0$ 两个儿子结点。其左儿子结点表示 $x[i]=1$ 的情形，仅当 $cw + w[i] \leqslant c$ 时进入左子树，递归地对左子树进行搜索。其右儿子结点表示 $x[i]=0$ 的情形。由于可行结点的右儿子结点总是可行的，故进入右子树时不需检查可行性。

函数 Backtrack 动态地生成问题的解空间树。在每个结点处算法花费 $O(1)$ 时间。子集树中结点个数为 $O(2^n)$，故 Backtrack 所需的计算时间为 $O(2^n)$。另外 Backtrack 还需要额外的 $O(n)$ 的递归栈空间。

具体算法可描述如下：

```
template < class Type >
class Loading {
   friend Type MaxLoading(Type [], Type, int);
   private:
      void Backtrack(int i);
      int n;          // 集装箱数
      Type * w,       // 集装箱重量数组
        c,            // 第一艘轮船的载重量
        cw,           // 当前载重量
        bestw;        // 当前最优载重量
};
```

```
template < class Type >
void Loading < Type > ::Backtrack(int i)
{// 搜索第 i 层结点
   if (i > n) {// 到达叶结点
      if (cw > bestw) bestw = cw;
      return;}
   // 搜索子树
   if (cw + w[i] < = c) {// x[i] = 1
      cw + = w[i];
      Backtrack(i + 1);
      cw - = w[i];}
   Backtrack(i + 1);// x[i] = 0
}
```

```
template < class Type >
Type MaxLoading(Type w[], Type c, int n)
```

```
}// 返回最优载重量
    Loading < Type > X;
    // 初始化 X
    X.w = w;
    X.c = c;
    X.n = n;
    X.bestw = 0;
    X.cw = 0;
    // 计算最优载重量
    X.Backtrack(1);
    return X.bestw;
}
```

3. 上界函数

对于前面所描述的算法 Backtrack,我们还可以引入一个上界函数,用于剪去不含最优解的子树,从而改进算法在平均情况下的运行效率。设 Z 是解空间树第 i 层上的当前扩展结点。cw 是当前载重量;bestw 是当前最优载重量;r 是剩余集装箱的重量,即 $r = \sum_{j=i+1}^{n} w_j$。定义上界函数为 cw + r。在以 Z 为根的子树中任一叶结点所相应的载重量均不超过 cw + r。因此,当 cw + $r \leqslant$ bestw 时,可将 Z 的右子树剪去。

在下面的改进算法中,引入类 Loading 的一个私有变量 r,用于计算上界函数。引入上界函数后,在达到一个叶结点时就不必再检查该叶结点是否优于当前最优解。因为上界函数使算法搜索到的每个叶结点都是迄今为止找到的最优解。虽然改进后的算法的计算时间复杂性仍为 $O(2^n)$,但在平均情况下改进后的算法检查的结点数较少。

改进后的算法可描述如下:

```
template < class Type >
class Loading {
    friend Type MaxLoading(Type [], Type, int);
    private:
      void Backtrack(int i);
      int n;         // 集装箱数
      Type * w,      // 集装箱重量数组
        c,           // 第一艘轮船的载重量
        cw,          // 当前载重量
        bestw,       // 当前最优载重量
        r;           // 剩余集装箱重量
};
```

```
template < class Type >
```

```
void Loading < Type > ::Backtrack(int i)
{// 搜索第 i 层结点
  if (i > n) {// 到达叶结点
    bestw = cw;
    return;}
  // 搜索子树
  r -= w[i];
  if (cw + w[i] <= c) {// x[i] = 1
    cw += w[i];
    Backtrack(i + 1);
    cw -= w[i];}
  if (cw + r > bestw) // x[i] = 0
    Backtrack(i + 1);
  r += w[i];
}
```

```
template < class Type >
Type MaxLoading(Type w[], Type c, int n)
{// 返回最优载重量
  Loading < Type > X;
  // 初始化 X
  X.w = w;
  X.c = c;
  X.n = n;
  X.bestw = 0;
  X.cw = 0;
  // 初始化 r
  X.r = 0;
  for (int i = 1; i <= n; i++)
    X.r += w[i];
  // 计算最优载重量
  X.Backtrack(1);
  return X.bestw;
}
```

4. 构造最优解

为了最终构造出达到最优值的最优解，必须在算法中记录与当前最优值相应的当前最优解。为此，在类 Loading 中增加两个私有数据成员 x 和 bestx。x 用于记录从根至当前结点的路径；，bestx 记录当前最优解。算法搜索到达一个叶结点处，就修正 bestx 的值。

进一步改进后的算法可描述如下：

```
template < class Type >
class Loading {
   friend Type MaxLoading(Type [], Type, int, int []);
   private:
      void Backtrack(int i);
      int n,          // 集装箱数
          * x,        // 当前解
          * bestx;    // 当前最优解
      Type * w,       // 集装箱重量数组
          c,          // 第一艘轮船的载重量
          cw,         // 当前载重量
          bestw,      // 当前最优载重量
          r;          // 剩余集装箱重量
};

template < class Type >
void Loading < Type > ::Backtrack(int i)
{// 搜索第 i 层结点
   if (i > n) {// 到达叶结点
      if (cw > bestw){ for (j = 1;j < = n;j ++) bestx[j] = x[j]; bestw = cw; }
      return;}
   // 搜索子树
   r -= w[i];
   if (cw + w[i] < = c) {// 搜索左子树
      x[i] = 1;
      cw += w[i];
      Backtrack(i + 1);
      cw -= w[i];}
   if (cw + r > bestw) {// 搜索右子树
      x[i] = 0;
      Backtrack(i + 1);}
   r += w[i];
}

template < class Type >
Type MaxLoading(Type w[], Type c, int n, int bestx[])
{// 返回最优载重量
   Loading < Type > X;
   // 初始化 X
   X.x = new int [n + 1];
   X.w = w;
   X.c = c;
   X.n = n;
```

```
    X.bestx = bestx;
    X.bestw = 0;
    X.cw = 0;
    // 初始化 r
    X.r = 0;
    for (int i = 1; i <= n; i++)
      X.r += w[i];
    X.Backtrack(1);
    delete [] X.x;
    return X.bestw;
}
```

由于 bestx 可能被更新 $O(2^n)$ 次,故改进后算法的计算时间复杂性为 $O(n2^n)$。

我们可以采用下面的两种策略之一使改进后的算法的计算时间复杂性减至 $O(2^n)$。

(1) 首先运行只计算最优值的算法,计算出最优装载量 W。由于该算法不记录最优解,故所需的计算时间为 $O(2^n)$。然后再运行改进后的算法 Backtrack,并在算法中将 bestw 置为 W。这样一来,在首次到达的叶结点处(即首次遇到 $i > n$ 时)终止算法。由此返回的 bestx 即为最优解。

(2) 另一种策略是在算法中动态地更新 bestx。在第 i 层的当前结点处,当前最优解由 $x[j], 1 \leqslant j < i$ 和 $bestx[j], i \leqslant j \leqslant n$ 所组成。每当算法回溯一层时,将 $x[i]$ 存入 $bestx[i]$。这样一来,在每个结点处更新 bestx 只需 $O(1)$ 时间,从而整个算法中更新 bestx 所需的时间为 $O(2^n)$。

5. 迭代回溯

由于数组 x 记录了解空间树中从根到当前扩展结点的路径,这些信息已包含了回溯法在回溯时所需要的信息。因此利用数组 x 所含的信息,可将上述回溯法表示成非递归的形式。这样,可进一步省去 $O(n)$ 的递归栈空间。解装载问题的非递归的迭代回溯法 MaxLoading 可描述如下:

```
template <class Type>
Type MaxLoading(Type w[], Type c, int n, int bestx[])
{ // 迭代回溯法
  // 返回最优载重量及其相应解
  // 初始化根结点
  int i = 1; // 当前层
  // x[1:i-1] 为当前路径
  int * x = new int [n + 1];
  Type bestw = 0, // 当前最优载重量
       cw = 0, // 当前载重量
       r = 0; // 剩余集装箱重量
  for (int j = 1; j <= n; j++)
    r += w[j];
```

```
// 搜索子树
while (true) {
  while (i <= n && cw + w[i] <= c) {
    // 进入左子树
    r -= w[i];
    cw += w[i];
    x[i] = 1;
    i++;
    }
  if (i > n) {// 到达叶结点
    for (int j = 1; j <= n; j++)
      bestx[j] = x[j];
    bestw = cw;}
  else {// 进入右子树
      r -= w[i];
      x[i] = 0;
      i++;}
  while (cw + r <= bestw) {
    // 剪枝回溯
    i--;
    while (i > 0 && !x[i]) {
      // 从右子树返回
      r += w[i];
      i--;
      }
    if (i == 0) {delete [] x;
           return bestw;}
    // 进入右子树
    x[i] = 0;
    cw -= w[i];
    i++;
    }
  }
}
```

算法 MaxLoading 所需的计算时间仍为 $O(2^n)$。

5.3 批处理作业调度

1. 问题描述

给定 n 个作业的集合 $J=(J_1,J_2,\cdots,J_n)$。每一个作业 J_i 都有两项任务要分别在 2 台机器上完成。每一个作业必须先由机器 1 处理，然后再由机器 2 处理。作业 J_i 需要机器 j 的处理时间

为 t_{ji}, $i = 1,2,\cdots,n$; $j = 1,2$。对于一个确定的作业调度，设 F_{ji} 是作业 i 在机器 j 上完成处理的时间。则所有作业在机器 2 上完成处理的时间和 $f = \sum_{i=1}^{n} F_{2i}$ 称为该作业调度的完成时间和。

批处理作业调度问题要求对于给定的 n 个作业，制定一个最佳作业调度方案，使其完成时间和达到最小。

批处理作业调度问题的一个常见例子是在计算机系统中完成一批 n 个作业，每个作业都要完成先计算，然后将计算结果打印输出这两项任务。计算任务由计算机的中央处理器完成，打印输出任务由打印机完成。因此在这种情形下，计算机的中央处理器是机器 1，打印机是机器 2。

对于批处理作业调度问题，可以证明，存在一个最佳作业调度使得在机器 1 和机器 2 上作业以相同次序完成。

例如，考虑如下 $n = 3$ 的实例：

t_{ji}	机器 1	机器 2
作业 1	2	1
作业 2	3	1
作业 3	2	3

这三个作业的 6 种可能的调度方案是 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2; 3,2,1；它们所相应的完成时间和分别是 19,18,20,21,19,19。显而易见，最佳调度方案是 1,3,2，其完成时间为 18。

2. 算法设计

对于批处理作业调度问题，由于我们要从 n 个作业的所有排列中找出有最小完成时间和的作业调度，所以批处理作业调度问题的解空间是一棵排列树。按照回溯法搜索排列树的算法框架，设开始时 x = [1,2,⋯,n] 是所给的 n 个作业，则相应的排列树由 x[1:n] 的所有排列构成。

解批处理作业调度问题的回溯算法 Backtrack 是类 Flowshop 的私有成员函数。函数 Flow 是 Flowshop 的友员。Flow 返回找到的最小完成时间和，bestx 返回相应的最佳作业调度。类 Flowshop 的其他成员记录解空间中结点信息，以减少传给函数 Backtrack 的参数。二维数组 M 是输入的作业处理时间。bestf 记录当前最小完成时间和，bestx 是相应的当前最佳作业调度。

在递归函数 Backtrack 中，当 $i > n$ 时，表示算法已搜索至一个叶结点，得到一个新的作业调度方案。此时算法适时更新当前最优值和相应的当前最佳作业调度。

当 $i < n$ 时，当前扩展结点位于排列树的第 $i-1$ 层。此时算法选择下一个要安排的作业，以深度优先的方式递归地对相应子树进行搜索。对于不满足上界约束的结点，则剪去相应的子树。

解批处理作业调度问题的回溯算法可描述如下：

```
class Flowshop {
```

```
    friend Flow(int * * , int, int [ ]);
    private:
      void Backtrack(int i);
      int * * M,          // 各作业所需的处理时间
        * x,              // 当前作业调度
        * bestx,          // 当前最优作业调度
        * f2,             // 机器 2 完成处理时间
        f1,               // 机器 1 完成处理时间
        f,                // 完成时间和
        bestf,            // 当前最优值
        n;                // 作业数
  };
```

```
void Flowshop::Backtrack(int i)
{
  if (i > n) {
    for (int j = 1; j <= n; j++)
      bestx[j] = x[j];
    bestf = f;
    }
  else
    for (int j = i; j <= n; j++) {
      f1 += M[x[j]][1];
      f2[i] = ((f2[i - 1] > f1) ? f2[i - 1]:f1) + M[x[j]][2];
      f += f2[i];
      if (f < bestf) {
        Swap(x[i], x[j]);
        Backtrack(i + 1);
        Swap(x[i], x[j]);
        }
      f1 -= M[x[j]][1];
      f -= f2[i];
      }
}
```

```
int Flow(int * * M, int n, int bestx[ ])
{
  int ub = 32767;
  Flowshop X;
  X.x = new int [n + 1];
  X.f2 = new int [n + 1];
  X.M = M;
```

```
    X.n = n;
    X.bestx = bestx;
    X.bestf = ub;
    X.f1 = 0;
    X.f = 0;
    for (int i = 0; i <= n; i++)
        X.f2[i] = 0, X.x[i] = i;
    X.Backtrack(1);
    delete [] X.x;
    delete [] X.f2;
    return X.bestf;
}
```

3. 算法效率

由于算法 Backtrack 在每一个结点处耗费 $O(1)$ 计算时间，故在最坏情况下，整个算法的计算时间复杂性为 $O(n!)$。

5.4 符号三角形问题

1. 问题描述

图 5-4 是由 14 个“+”号和 14 个“-”号组成的符号三角形。2 个同号下面都是“+”号，2 个异号下面都是“-”号。

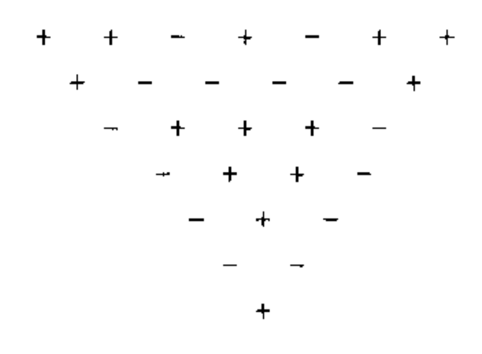


图 5-4 符号三角形

在一般情况下，符号三角形的第一行有 n 个符号。符号三角形问题要求对于给定的 n，计算有多少个不同的符号三角形，使其所含的“+”和“-”的个数相同。

2. 算法设计

对于符号三角形问题，我们用 n 元组 x[1:n] 表示符号三角形的第一行的 n 个符号。当 x[i] = 1 时，表示符号三角形的第一行的第 i 个符号为“+”号；当 x[i] = 0 时，表示符号三角形的第一行的第 i 个符号为“-”号；$1 \leqslant i \leqslant n$。由于 x[$i$] 是二值的，所以在用回溯法解符号三角形问题时，可以用一棵完全二叉树来表示其解空间。在符号三角形的第一行的前 i 个符号

x[1:i] 确定后，就确定了一个由 $i*(i+1)/2$ 个符号组成的符号三角形。下一步确定了 x[$i+1$] 的值后，只要在前面已确定的符号三角形的右边加一条边，就可以扩展为 x[1:$i+1$] 所相应的符号三角形。最终由 x[1:n] 所确定的符号三角形中包含的"+"号个数与"-"号个数同为 $n*(n+1)/4$。因此在回溯搜索过程中可用当前符号三角形所包含的"+"号个数与"-"号个数均不超过 $n*(n+1)/4$ 作为可行性约束，用于剪去不满足约束的子树。对于给定的 n，当 $n*(n+1)/2$ 为奇数时，显然不存在所包含的"+"号个数与"-"号个数相同的符号三角形。这种情况可以通过简单的判断加以处理。

在下面所给出的解符号三角形问题的回溯法描述中，递归函数 Backtrack(1) 实现对整个解空间的回溯搜索。Backtrack(i) 搜索解空间中第 i 层子树。函数 Backtrack 是类 Triangle 的成员。类 Triangle 的其他成员记录解空间中结点信息，以减少传给函数 Backtrack 的参数。sum 记录当前已找到的"+"号个数与"-"号个数相同的符号三角形数。函数 Compute 负责类 Triangle 的私有变量的初始化。

在函数 Backtrack 中，当 $i > n$ 时，表示算法已搜索至一个叶结点，得到一个新的"+"号个数与"-"号个数相同的符号三角形，因此当前已找到符号三角形数 sum 增 1。

当 $i \leqslant n$ 时，当前扩展结点 Z 是解空间中的一个内部结点。该结点有 x[i] = 1 和 x[i] = 0 共 2 个儿子结点。对当前扩展结点 Z 的每一个儿子结点，计算其相应的符号三角形中"+"号个数 count 与"-"号个数，并以深度优先的方式递归地对可行子树进行搜索，或剪去不可行子树。

解符号三角形问题的回溯算法可描述如下：

```
class Triangle {
    friend int Compute(int);
    private:
        void Backtrack(int t);
        int n,          // 第一行的符号个数
            half,       // n * (n + 1)/4
            count,      // 当前"+"号个数
            * * p;      // 符号三角形矩阵
        long sum;       // 已找到的符号三角形数
};
```

```
void Triangle::Backtrack(int t)
{
    if ((count > half) || (t * (t - 1)/2 - count > half)) return;
    if (t > n) sum ++;
      else
        for (int i = 0;i < 2;i ++) {
          p[1][t] = i;
          count += i;
          for (int j = 2;j <= t;j ++) {
            p[j][t - j + 1] = p[j - 1][t - j + 1]^p[j - 1][t - j + 2];
            count += p[j][t - j + 1];
```

```
        }
        Backtrack(t + 1);
        for (int j = 2;j <= t;j ++)
          count -= p[j][t - j + 1];
        count -= i;
      }
  }
```

```
int Compute(int n)
{
    Triangle X;
    X.n = n;
    X.count = 0;
    X.sum = 0;
    X.half = n * (n + 1)/2;
    if (X.half%2 == 1) return 0;
    X.half = X.half/2;
    int * * p = new int * [n + 1];
    for (int i = 0; i <= n; i ++)
        p[i] = new int [n + 1];
    for (int i = 0; i <= n; i ++)
    for (int j = 0; j <= n; j ++) p[i][j] = 0;
    X.p = p;
    X.Backtrack(1);
    return X.sum;
}
```

3. 算法效率

由于计算可行性约束需要 $O(n)$ 时间,在最坏情况下有 $O(2^n)$ 个结点需要计算可行性约束,故解符号三角形问题的回溯算法 Backtrack 所需的计算时间为 $O(n2^n)$。

5.5 n 后问题

1. 问题描述

n 后问题要求在一个 $n \times n$ 格的棋盘上放置 n 个皇后,使得他们彼此不受攻击。按照国际象棋的规则,一个皇后可以攻击与之处在同一行或同一列或同一斜线上的其他任何棋子。因此,n 后问题等价于要求在一个 $n \times n$ 格的棋盘上放置 n 个皇后,使得任何 2 个皇后不能被放在同一行或同一列或同一斜线上。

2. 算法设计

对于 n 后问题,我们用 n 元组 x[1:n] 表示它的解。其中,x[i] 表示皇后 i 放在棋盘的第 i

行的第 x[i]列。由于不允许将任何2个皇后放在同一列上,所以解向量中的诸 x[i]互不相同。在这里,任何2个皇后不能放在同一斜线上是问题的隐约束。对于一般的 n 后问题,这一隐约束条件可以化成显约束的形式。如果将 $n \times n$ 格的棋盘看作是一个二维方阵,其行号从上到下,列号从左到右依次编号为 $1,2,\cdots,n$,那么,从左上角到右下角的主对角线及其平行线(即斜率为 -1 的各斜线)上,元素的2个下标值的差(行号 - 列号)值相等。同理,斜率为 +1 的每一条斜线上,元素的2个下标值的和(行号 + 列号)值相等。因此,若2个皇后放置的位置分别是(i,j)和(k,l),且 $i-j=k-l$ 或 $i+j=k+l$,则说明这2个皇后处于同一斜线上。以上2个方程分别等价于 $i-k=j-l$ 和 $i-k=l-j$。由此可知,只要 $|i-k|=|j-l|$ 成立,就表明这2个皇后位于同一条斜线上。于是,问题的隐约束化成了显约束。据此我们可以设计一个函数 Place 来测试若将皇后 k 放在 x[k]列是否与前面已放置的 $k-1$ 个皇后都不在同一列,而且都不在同一斜线上。

用回溯法解 n后问题时,可以用一棵完全 n 叉树来表示其解空间。用可行性约束函数 Place 可剪去不满足行、列和斜线约束的子树。

在下面所给出的解 n 后问题的回溯法描述中,递归函数 Backtrack(1)实现对整个解空间的回溯搜索。Backtrack(i)搜索解空间中第 i 层子树。函数 Backtrack 是类 Queen 的成员。类 Queen 的其他成员记录解空间中结点信息,以减少传给函数 Backtrack 的参数。sum 记录当前已找到的可行方案数。函数 nQueen 负责类 Queen 的私有变量的初始化。

在函数 Backtrack 中,当 $i>n$ 时,表示算法已搜索至一个叶结点,得到一个新的 n 皇后互不攻击放置方案,因此当前已找到的可行方案数 sum 增 1。

当 $i \leqslant n$ 时,当前扩展结点 Z 是解空间中的一个内部结点。该结点有 x[i] $=1,2,\cdots,n$ 共 n 个儿子结点。对当前扩展结点 Z 的每一个儿子结点,由函数 Place 检查其可行性,并以深度优先的方式递归地对可行子树进行搜索,或剪去不可行子树。

解 n 后问题的回溯算法可描述如下:

```
class Queen {
   friend int nQueen(int);
   private:
      bool Place(int k);
      void Backtrack(int t);
      int n, // 皇后个数
          * x; // 当前解
      long sum; // 当前已找到的可行方案数
};
```

```
bool Queen::Place(int k)
{
   for (int j = 1;j < k;j++)
      if ((abs(k - j) == abs(x[j] - x[k])) || (x[j] == x[k]))return false;
   return true;
}
```

```
void Queen::Backtrack(int t)
{
   if (t > n) sum ++;
     else
       for (int i = 1;i < = n;i ++) {
         x[t] = i;
         if (Place(t)) Backtrack(t + 1);
       }
}
```

```
int nQueen(int n)
{
   Queen X;
   // 初始化 X
   X.n = n;
   X.sum = 0;
   int * p = new int [n + 1];
   for (int i = 0; i < = n; i ++)
     p[i] = 0;
   X.x = p;
   X.Backtrack(1);
   delete [] p;
   return X.sum;
}
```

3. 迭代回溯

由于数组 x 记录了解空间树中从根到当前扩展结点的路径，这些信息已包含了回溯法在回溯时所需要的信息。因此利用数组 x 所含的信息，可将上述回溯法表示成非递归的形式。这样一来，可省去 $O(n)$ 的递归栈空间。解 n 后问题的非递归迭代回溯法 Backtrack 可描述如下：

```
class Queen {
   friend int nQueen(int);
   private:
     bool Place(int k);
     void Backtrack(void);
     int n,        // 皇后个数
        * x;       // 当前解
     long sum;     // 当前已找到的可行方案数
};
```

```
bool Queen::Place(int k)
{
  for (int j = 1;j < k;j++)
    if ((abs(k - j) == abs(x[j] - x[k])) || (x[j] == x[k])) return false;
  return true;
}
```

```
void Queen::Backtrack(void)
{
  x[1] = 0;
  int k = 1;
  while (k > 0) {
    x[k] += 1;
    while ((x[k] <= n)&& !(Place(k))) x[k] += 1;
      if (x[k] <= n)
        if (k == n) sum++;
          else {
            k++;
            x[k] = 0;
            }
        else k--;
    }
}
```

```
int nQueen(int n)
{
  Queen X;
  // 初始化 X
  X.n = n;
  X.sum = 0;
  int *p = new int [n + 1];
  for (int i = 0; i <= n; i++)
    p[i] = 0;
  X.x = p;
  X.Backtrack();
  delete [] p;
  return X.sum;
}
```

5.6　0-1 背包问题

1. 算法描述

0-1 背包问题也是一个子集选取问题。在一般情况下，0-1 背包问题是 NP 难的。适合于用子集树表示 0-1 背包问题的解空间。解 0-1 背包问题的回溯法与 5.2 节中讨论的解装载问题的回溯法十分相似。在搜索解空间树时，只要其左儿子结点是一个可行结点，搜索就进入其左子树。在右子树中有可能包含最优解时才进入右子树搜索。否则将右子树剪去。这个任务由上界函数来完成。设 r 是当前尚未考虑的剩余物品价值总和；cp 是当前价值；bestp 是当前最优价值。则当 $cp + r \leqslant bestp$ 时，可剪去右子树。计算右子树中解的上界的一个更好的方法是将剩余物品依其单位重量价值排序，然后依次装入物品，直至装不下时，再装入该物品的一部分而装满背包。由此得到的价值是右子树中解的一个上界。

例如，对于 0-1 背包问题的一个实例，$n = 4, c = 7, p = [9,10,7,4], w = [3,5,2,1]$。这 4 个物品的单位重量价值分别为[3,2,3.5,4]。以物品单位重量价值的递减序装入物品。首先装入物品 4，然后装入物品 3 和 1。装入这 3 个物品后，剩余的背包容量为 1，只能装入 0.2 的物品 2。由此得到一个解为 $x = [1,0.2,1,1]$，其相应的价值为 22。尽管这个解不是一个可行解，但可以证明其价值是最优值的一个上界。因此，对于这个实例，最优值不超过 22。

为了便于计算上界函数，可先将物品依其单位重量价值从大到小排序，此后只要按顺序考察各物品即可。在实现时，由函数 Bound 来计算在当前结点处的上界。它是类 Knap 的私有成员。Knap 的其他成员记录解空间树中的结点信息，以减少函数参数的传递以及递归调用时所需的栈空间。在解空间树的当前扩展结点处，仅当要进入右子树时才计算上界函数 Bound，以判断是否可以将右子树剪去。进入左子树时不需计算上界，因为其上界与其父结点的上界相同。

在调用函数 Knapsack 之前，需先将各物品依其单位重量价值从大到小排序。为此目的，我们定义了类 Object。其中，<= 运算符与通常的定义相反，其目的是为了方便调用已有的排序算法。在通常情况下，排序算法将待排序元素从小到大排列。

解 0-1 背包问题的回溯算法可描述如下：

```
template < class Typew, class Typep >
class Knap {
   friend Typep Knapsack(Typep * , Typew * , Typew, int);
   private:
      Typep Bound(int i);
      void Backtrack(int i);
      Typew c;           // 背包容量
      int n;             // 物品数
      Typew * w;         // 物品重量数组
      Typep * p;         // 物品价值数组
      Typew cw;          // 当前重量
      Typep cp;          // 当前价值
```

```
    Typep bestp;        // 当前最优价值
};

template < class Typew, class Typep >
Typep Knap < Typew, Typep > ::Bound(int i)
{// 计算上界
  Typew cleft = c - cw; // 剩余容量
  Typep b = cp;
  // 以物品单位重量价值递减序装入物品
  while (i < = n && w[i] < = cleft) {
    cleft - = w[i];
    b + = p[i];
    i + +;
    }
  // 装满背包
  if (i < = n) b + = p[i]/w[i] * cleft;
  return b;
}

template < class Typew, class Typep >
void Knap < Typew, Typep > ::Backtrack(int i)
{
  if (i > n) {
    bestp = cp;
    return;}
  if (cw + w[i] < = c) {// x[i] = 1
    cw + = w[i];
    cp + = p[i];
    Backtrack(i + 1);
    cw - = w[i];
    cp - = p[i];}
  if (Bound(i + 1) > bestp) // x[i] = 0
    Backtrack(i + 1);
}

class Object {
  friend int Knapsack(int * , int   * , int, int);
  public:
    int operator < = (Object a) const
    {return (d > = a.d);}
  private:
```

```
    int ID;
    float d;
};

template < class Typew, class Typep >
Typep Knapsack(Typep p[], Typew w[], Typew c, int n)
{
  // 为 Knap::Backtrack 初始化
  Typew W = 0;
  Typep P = 0;
  Object * Q = new Object [n];
  for (int i = 1; i < = n; i ++) {
    Q[i - 1].ID = i;
    Q[i - 1].d = 1.0 * p[i]/w[i];
    P + = p[i];
    W + = w[i];
    }
  if (W < = c) return P; // 装入所有物品
  // 依物品单位重量价值排序
  Sort(Q,n);
  Knap < Typew, Typep > K;
  K.p = new Typep [n + 1];
  K.w = new Typew [n + 1];
  for (int i = 1; i < = n; i ++) {
    K.p[i] = p[Q[i - 1].ID];
    K.w[i] = w[Q[i - 1].ID];
    }
  K.cp = 0;
  K.cw = 0;
  K.c = c;
  K.n = n;
  K.bestp = 0;
  // 回溯搜索
  K.Backtrack(1);
  delete [] Q;
  delete [] K.w;
  delete [] K.p;
  return K.bestp;
}
```

2. 算法效率

由于计算上界函数 Bound 需要 $O(n)$ 时间，在最坏情况下有 $O(2^n)$ 个右儿子结点需要计算上界函数，故解 0-1 背包问题的回溯算法 Backtrack 所需的计算时间为 $O(n2^n)$。

5.7 最大团问题

1. 问题描述

给定一个无向图 $G=(V,E)$。如果 $U\subseteq V$，且对任意 $u,v\in U$ 有 $(u,v)\in E$，则称 U 是 G 的一个完全子图。G 的完全子图 U 是 G 的一个团当且仅当 U 不包含在 G 的更大的完全子图中。G 的最大团是指 G 中所含顶点数最多的团。

在图 5-5 中的无向图 G 中，子集{1,2}是 G 的一个大小为 2 的完全子图。这个完全子图不是一个团，因为它包含于 G 的更大的完全子图{1,2,5}之中。{1,2,5}是 G 的一个最大团。{1,4,5}和{2,3,5}也是 G 的最大团。

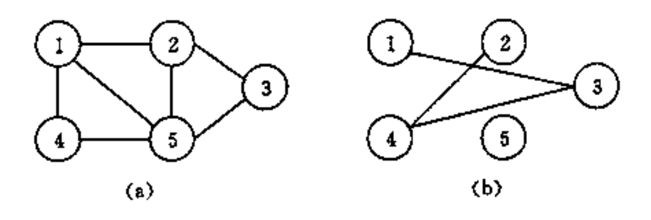


图 5-5　无向图 G 和 G 的补图 $\overline{G}$

如果 $U\subseteq V$ 且对任意 $u,v\in U$ 有 $(u,v)\notin E$，则称 U 是 G 的一个空子图。G 的空子图 U 是 G 的一个独立集当且仅当 U 不包含在 G 的更大的空子图中。G 的最大独立集是 G 中所含顶点数最多的独立集。

对于任一无向图 $G=(V,E)$ 其补图 $\overline{G}=(V1,E1)$ 定义为：$V1=V$，且 $(u,v)\in E1$ 当且仅当 $(u,v)\notin E$。

图 5-5(a) 和图 5-5(b) 中的两个无向图互为补图。{2,4}是 G 的一个空子图，同时也是 G 的一个最大独立集。虽然{1,2}是 $\overline{G}$ 的空子图，但它不是 $\overline{G}$ 的独立集，因为它包含在 $\overline{G}$ 的空子图{1,2,5}中。{1,2,5}是 $\overline{G}$ 的最大独立集。

我们注意到，如果 U 是 G 的一个完全子图，则它是 $\overline{G}$ 的一个空子图，反之亦然。因此，G 的团与 $\overline{G}$ 的独立集之间存在一一对应关系。特别的，U 是 G 的最大团当且仅当 U 是 $\overline{G}$ 的最大独立集。

2. 算法设计

无向图 G 的最大团和最大独立集问题都可以用回溯法在 $O(n2^n)$ 时间内解决。图 G 的最大团和最大独立集问题都可以看作是图 G 的顶点集 V 的子集选取问题。因此可以用子集树表示问题的解空间。解最大团问题的回溯法与解装载问题的回溯法十分相似。设当前扩展结点 Z 位于解空间树的第 i 层。在进入左子树前，必须确认从顶点 i 到已选入的顶点集中每一个顶点都有边相连。在进入右子树前，必须确认还有足够多的可选择顶点使得算法有可能在右子树中找到更大的团。

在具体实现时，用邻接矩阵表示图 G。函数 Backtrack 是类 Clique 的私有成员函数，而函数 MaxClique 负责有关变量的初始化以及调用 Backtrack 进行搜索。Backtrack 具体实施对解空间的回溯搜索。函数 MaxClique 返回最大团的大小；整型数组 v 返回所找到的最大团。v[i] = 1 当且仅当顶点 i 属于找到的最大团。

解最大团问题的回溯算法可描述如下：

```
class Clique {
   friend MaxClique(int * *, int [], int);
   private:
      void Backtrack(int i);
      int    * * a,      // 图 G 的邻接矩阵
             n,          // 图 G 的顶点数
             * x,        // 当前解
             * bestx,    // 当前最优解
             cn;         // 当前顶点数
             bestn,      // 当前最大顶点数
};
```

```
void Clique::Backtrack(int i)
{// 计算最大团
   if (i > n) {
      for (int j = 1; j <= n; j++)
         bestx[j] = x[j];
      bestn = cn;
      return;}
   // 检查顶点 i 与当前团的连接
   int OK = 1;
   for (int j = 1; j < i; j++)
      if (x[j] && a[i][j] == 0) {
         // i 与 j 不相连
         OK = 0;
         break;}
   if (OK) {// x[i] = 1
      x[i] = 1;
      cn++;
      Backtrack(i + 1);
      x[i] = 0;
      cn--;}
   if (cn + n - i > bestn) {
      x[i] = 0;
      Backtrack(i + 1);}
}
```

```
int MaxClique(int * * a, int v[], int n)
{
    Clique Y;
    // 初始化 Y
    Y.x = new int[n + 1];
    Y.a = a;
    Y.n = n;
    Y.cn = 0;
    Y.bestn = 0;
    Y.bestx = v;
    Y.Backtrack(1);
    delete [] Y.x;
    return Y.bestn;
}
```

3. 算法效率

解最大团问题的回溯算法 Backtrack 所需的计算时间显然为 $O(n2^n)$。

5.8 图的 m 着色问题

1. 问题描述

给定一个无向连通图 G 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。试问是否有使得 G 中任何一条边的2个顶点着有不同颜色的着色法。这个问题就是一个图的 m 可着色判定问题。若一个图最少需要 m 种颜色才能使图中任何一条边连接的2个顶点着有不同颜色，则称这个数 m 为该图的色数。求一个图的色数 m 的问题称为图的 m 可着色优化问题。

如果一个图的所有顶点和边都能用某种方式画在一个平面上且没有任何两边相交，则称这个图是可平面图。著名的平面图的四色猜想是图的 m 可着色性判定问题的一个特殊情形。这个猜想可表述为：在一个平面或球面上的任何地图能够只用 4 种颜色来着色，使得相邻的国家在地图上着有不同颜色。这里假设每个国家在地图上必须是一个单连通域，还假设 2 个国家相邻是指这 2 个国家有一段长度不为 0 的公共边界，而不是只有一个公共点。任何一个这样的地图很容易用一个平面图来表示。地图上的每一个区域相应于平面图中一个顶点。若在地图上 2 个区域是相邻的，则它们在平面图中相应的 2 个顶点之间有一条边相连。图 5-6 是一个有 5 个区域的地图及其相应的平面图。这个地图需要 4 种颜色来着色。

很早以前就已知道用 5 种颜色就足以为任何一个地图着色。另一方面又一直没找到一个需要 4 种以上颜色才能着色的地图，由此引出了四色猜想。这个猜想直到 1976 年才由三个美国人依靠计算机的帮助做出了证明：任何平面图都是可以 4 着色的。四色猜想从此变成了四色

定理。

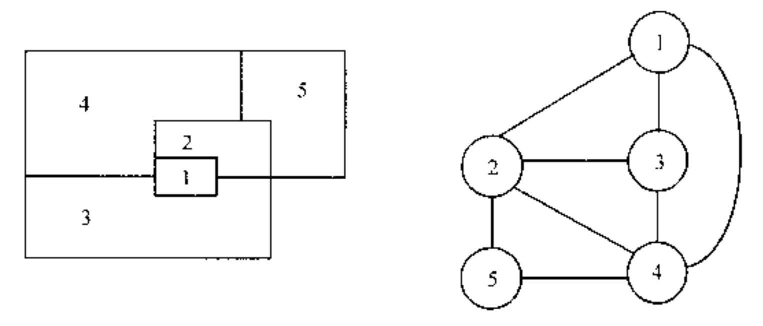


图 5-6　地图及其相应的平面图

2. 算法设计

在这一节中,我们要讨论一般连通图的可着色性问题,而不仅限于平面图。我们感兴趣的是,给定了一个图 $G = (V, E)$ 和 m 种颜色,如果这个图不是 m 可着色的就给出否定回答;如果这个图是 m 可着色的,则要找出所有不同的着色法。要解决这个问题,除了用回溯法外,目前还没有什么更好的方法。

下面根据回溯法的递归描述框架 Backtrack 来设计找一个图的 m 着色法的算法。算法中用图的邻接矩阵 a 来表示一个无向连通图 $G = (V, E)$。若 (i, j) 属于图 $G = (V, E)$ 的边集 E,则 a[i][j] = 1,否则 a[i][j] = 0。用整数 $1, 2, \cdots, m$ 来表示 m 种不同的颜色。顶点 i 所着的颜色用 x[i] 来表示。因此,该问题的解向量可以表示为 n 元组 x[1:n]。问题的解空间可表示为一棵高度为 $n+1$ 的完全 m 叉树。解空间树的第 $i(1 \leqslant i \leqslant n)$ 层中每一结点都有 m 个儿子,每个儿子相应于 x[i] 的 m 个可能的着色之一。第 $n+1$ 层结点均为叶结点。图 5-7 是 $n = 3$ 和 $m = 3$ 时问题的解空间树。

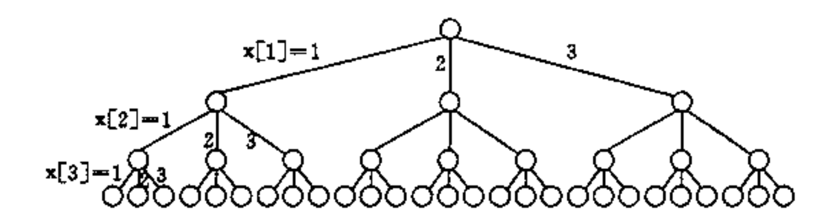


图 5-7　$n = 3$ 和 $m = 3$ 时的解空间树

在下面所给出的解图的 m 可着色问题的回溯法描述中,递归函数 Backtrack(1) 实现对整个解空间的回溯搜索。Backtrack(i) 搜索解空间中第 i 层子树。函数 Backtrack 是类 Color 的成员。类 Color 的其他成员记录解空间中结点信息,以减少传给函数 Backtrack 的参数。sum 记录当前已找到的可 m 着色方案数。函数 mColoring 负责类 Color 的私有变量的初始化。

在函数 Backtrack 中,当 $i > n$ 时,表示算法已搜索至一个叶结点,得到一个新的 m 着色方案,因此当前已找到的可 m 着色方案数 sum 增 1。

当 $i \leqslant n$ 时,当前扩展结点 Z 是解空间中的一个内部结点。该结点有 x[i] = $1, 2, \cdots, m$ 共 m 个儿子结点。对当前扩展结点 Z 的每一个儿子结点,由函数 Ok 检查其可行性,并以深度优先的方式递归地对可行子树进行搜索,或剪去不可行子树。

解图的 m 可着色问题的回溯算法可描述如下:

```
class Color {
```

```
    friend int mColoring(int,int,int * * );
    private:
      bool Ok(int k);
      void Backtrack(int t);
      int    n,              // 图的顶点数
             m,              // 可用颜色数
             * * a;          // 图的邻接矩阵
             * x,            // 当前解
             long sum;       // 当前已找到的可 m 着色方案数
};
```

```
bool Color::Ok(int k)
{// 检查颜色可用性
   for (int j = 1;j < = n;j ++)
     if ((a[k][j] = = 1)&&(x[j] = = x[k])) return false;
   return true;
}
```

```
void Color::Backtrack(int t)
{
   if (t > n) {
   sum ++;
   for (int i = 1; i < = n; i ++)
     cout < < x[i] < < ' ';
   cout < < endl;
   }
     else
       for (int i = 1;i < = m;i ++) {
         x[t] = i;
         if (Ok(t)) Backtrack(t + 1);
       }
}
```

```
int mColoring(int n, int m, int * * a)
{
   Color X;
   // 初始化 X
   X.n = n;
   X.m = m;
   X.a = a;
   X.sum = 0;
```

```
    int * p = new int [n + 1];
    for (int i = 0; i <= n; i++)
      p[i] = 0;
    X.x = p;
    X.Backtrack(1);
    delete [] p;
    return X.sum;
}
```

3. 算法效率

解图的 m 可着色问题的回溯算法的计算时间上界可以通过计算解空间树中内结点个数来估计。图的 m 可着色问题的解空间树中内结点个数是 $\sum_{i=0}^{n-1} m^i$。对于每一个内结点，在最坏情况下，函数 Ok 检查当前扩展结点的每一个儿子所相应的颜色的可用性需耗时 $O(mn)$。因此，回溯算法总的时间耗费是 $\sum_{i=0}^{n-1} m^i(mn) = nm(m^n - 1)/(m - 1) = O(nm^n)$。

5.9 旅行售货员问题

1. 算法描述

旅行售货员问题的解空间是一棵排列树。对于排列树的回溯搜索与生成 $1,2,\cdots,n$ 的所有排列的递归算法 Perm 类似。设开始时 $x = [1,2,\cdots,n]$，则相应的排列树由 $x[1:n]$ 的所有排列构成。

找旅行售货员回路的回溯算法 Backtrack 是类 Traveling 的私有成员函数，TSP 是 Traveling 的友员。TSP(v) 返回旅行售货员回路最小费用。整型数组 v 返回相应的回路。如果所给的图 G 不含旅行售货员回路，则返回 NoEdge。函数 TSP 所做的工作主要是为调用 Backtrack 所需的变量初始化。由 TSP 调用 Backtrack(2) 搜索整个解空间。

在递归函数 Backtrack 中，当 $i = n$ 时，当前扩展结点是排列树的叶结点的父结点。此时算法检测图 G 是否存在一条从顶点 $x[n-1]$ 到顶点 $x[n]$ 的边和一条从顶点 $x[n]$ 到顶点 1 的边。如果这两条边都存在，则找到一条旅行售货员回路。此时，算法还需判断这条回路的费用是否优于已找到的当前最优回路的费用 bestc。如果是，则必须更新当前最优值 bestc 和当前最优解 bestx。

当 $i < n$ 时，当前扩展结点位于排列树的第 $i-1$ 层。图 G 中存在从顶点 $x[i-1]$ 到顶点 $x[i]$ 的边时，$x[1:i]$ 构成图 G 的一条路径，且当 $x[1:i]$ 的费用小于当前最优值时算法进入排列树的第 i 层，否则将剪去相应的子树。算法中用变量 cc 记录当前路径 $x[1:i]$ 的费用。

解旅行售货员问题的回溯算法可描述如下：

```
template <class Type>
class Traveling {
```

```
    friend Type TSP(int * * , int [], int, Type);
  private:
    void Backtrack(int i);
    int n,        // 图 G 的顶点数
        * x,      // 当前解
        * bestx;  // 当前最优解
  Type * * a,     // 图 G 的邻接矩阵
        cc,       // 当前费用
        bestc,    // 当前最优值
        NoEdge;   // 无边标记
  };
```

```
template < class Type >
void Traveling < Type > ::Backtrack(int i)
{
  if (i = = n) {
    if (a[x[n - 1]][x[n]] ! = NoEdge &&
      a[x[n]][1] ! = NoEdge &&
      (cc + a[x[n - 1]][x[n]] + a[x[n]][1] < bestc ||
      bestc = = NoEdge)) {
      for (int j = 1; j < = n; j ++)
        bestx[j] = x[j];
      bestc = cc + a[x[n - 1]][x[n]] + a[x[n]][1];}
    }
  else {
    for (int j = i; j < = n; j ++)
      // 是否可进入 x[j] 子树?
      if (a[x[i - 1]][x[j]] ! = NoEdge &&
          (cc + a[x[i - 1]][x[i]] < bestc ||
          bestc = = NoEdge)) {
        // 搜索子树
        Swap(x[i], x[j]);
        cc + = a[x[i - 1]][x[i]];
        Backtrack(i + 1);
        cc - = a[x[i - 1]][x[i]];
        Swap(x[i], x[j]);}
    }
}
```

```
template < class Type >
Type TSP(Type * * a, int v[], int n, Type NoEdge)
{
```

```
Traveling < Type > Y;
// 初始化 Y
Y.x = new int [n + 1];
// 置 x 为单位排列
for (int i = 1; i < = n; i ++)
Y.x[i] = i;
Y.a = a;
Y.n = n;
Y.beste = NoEdge;
Y.bestx = v;
Y.cc = 0;
Y.NoEdge = NoEdge;
// 搜索 x[2:n] 的全排列
Y.Backtrack(2);
delete [] Y.x;
return Y.beste;
```

2. 算法效率

如果不考虑更新 bestx 所需的计算时间，则 Backtrack 需要 $O((n-1)!)$ 计算时间。由于算法 Backtrack 在最坏情况下可能需要更新 $O((n-1)!)$ 次当前最优解 bestx，每次更新 bestx 需 $O(n)$ 计算时间，从而整个算法的计算时间复杂性为 $O(n!)$。

5.10 圆排列问题

1. 问题描述

给定 n 个大小不等的圆 $c_1, c_2, \cdots, c_n$，现要将这 n 个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。例如，当 $n = 3$，且所给的 3 个圆的半径分别为 1,1,2 时，这 3 个圆的最小长度的圆排列如图 5-8 所示，其最小长度为 $2 + 4\sqrt{2}$。

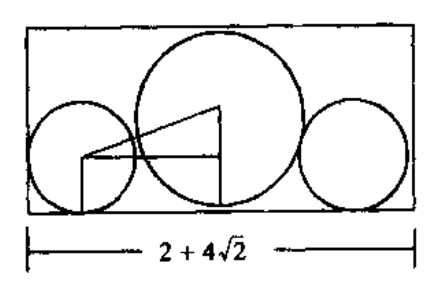


图 5-8　最小长度圆排列

2. 算法设计

由于我们要从 n 个圆的所有排列中找出有最小长度的圆排列，所以圆排列问题的解空间是一棵排列树。按照回溯法搜索排列树的算法框架，设开始时 $a = [r_1, r_2, \cdots, r_n]$ 是所给的 n 个圆的半径，则相应的排列树由 $a[1:n]$ 的所有排列构成。

解圆排列问题的回溯算法 Backtrack 是类 Circle 的私有成员函数，CirclePerm 是 Circle 的友员。CirclePerm(n, a) 返回找到的最小圆排列长度。初始时数组 a 是输入的 n 个圆的半径，计算结束后返回相应于最优解的圆排列。函数 Center 也是类 Circle 的私有成员函数，用于计算当前所选择的圆在当前圆排列中圆心的横坐标。函数 Compute 是类 Circle 的另一个私有成员函数，用于计算当前圆排列的长度。类 Circle 的私有变量 min 用于记录当前最小圆排列的长度；数组 r 表示当前圆排列；数组 x 则记录当前圆排列中各圆的圆心横坐标。算法中约定在当前圆排列中排在第一个的圆的圆心横坐标为 0。

在递归函数 Backtrack 中，当 $i > n$ 时，表示算法已搜索至一个叶结点，得到一个新的圆排列方案。此时算法调用函数 Compute 计算当前圆排列的长度，适时更新当前最优值。

当 $i < n$ 时，当前扩展结点位于排列树的第 $i-1$ 层。此时算法选择下一个要排列的圆，并计算相应的下界函数。在满足下界约束的结点处，以深度优先的方式递归地对相应子树进行搜索。对于不满足下界约束的结点，则剪去相应的子树。

解圆排列问题的回溯算法可描述如下：

```
class Circle {
   friend float CirclePerm(int, float * );
   private:
      float Center(int t);
      void Compute(void);
      void Backtrack(int t);
      float min,      // 当前最优值
            * x,      // 当前圆排列圆心横坐标
            * r;      // 当前圆排列
      int n;          // 待排列圆的个数
};
```

```
float Circle::Center(int t)
{// 计算当前所选择圆的圆心横坐标
   float temp = 0;
   for (int j = 1;j < t;j ++) {
      float valuex = x[j] + 2.0 * sqrt(r[t] * r[j]);
      if (valuex > temp) temp = valuex;
      }
   return temp;
}
```

```
void Circle::Compute(void)
{// 计算当前圆排列的长度
   float low = 0,
     high = 0;
   for (int i = 1;i <= n;i++) {
     if (x[i] - r[i] < low) low = x[i] - r[i];
     if (x[i] + r[i] > high) high = x[i] + r[i];
     }
   if (high - low < min) min = high - low;
}
```

```
void Circle::Backtrack(int t)
{
   if (t > n) Compute();
   else
   for (int j = t; j <= n; j++) {
        Swap(r[t], r[j]);
        float centerx = Center(t);
        if (centerx + r[t] + r[1] < min) {// 下界约束
          x[t] = centerx;
          Backtrack(t + 1);
          }
        Swap(r[t], r[j]);
        }
}
```

```
float CirclePerm(int n, float  *a)
{
   Circle X;
   X.n = n;
   X.r = a;
   X.min = 100000;
   float *x = new float [n + 1];
   X.x = x;
   X.Backtrack(1);
   delete [] x;
   return X.min;
}
```

3. 算法效率

如果不考虑计算当前圆排列中各圆的圆心横坐标和计算当前圆排列长度所需的计算时间,则 Backtrack 需要 $O(n!)$ 计算时间。由于算法 Backtrack 在最坏情况下可能需要计算 $O(n!)$ 次当前圆排列长度,每次计算需 $O(n)$ 计算时间,从而整个算法的计算时间复杂性为 $O((n+1)!)$。

上述算法尚有许多改进的余地。例如,像 $1,2,\cdots,n-1,n$ 和 $n,n-1,\cdots,2,1$ 这种互为镜像的排列具有相同的圆排列长度,只计算一个就够了。这样一来,可减少约一半的计算量。另一方面,如果所给的 n 个圆中有 k 个圆有相同的半径,则这 k 个圆产生的 $k!$ 个完全相同的圆排列,只计算一个就够了。上述算法的这些改进,留作练习。

5.11 电路板排列问题

1. 问题描述

电路板排列问题是大规模电子系统设计中提出的一个实际问题。该问题的经典提法是:将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B=\{1,2,\cdots,n\}$ 是 n 块电路板的集合。集合 $L=\{N_1,N_2,\cdots,N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集,且 N_i 中的电路板用同一根导线连接在一起。

例如,设 $n=8,m=5$。给定 n 块电路板及其 m 个连接块如下:

$B=\{1,2,3,4,5,6,7,8\}$; $L=\{N_1,N_2,N_3,N_4,N_5\}$;

$N_1=\{4,5,6\}$; $N_2=\{2,3\}$; $N_3=\{1,3\}$; $N_4=\{3,6\}$; $N_5=\{7,8\}$。

这 8 块电路板的一个可能的排列如图 5-9 所示。

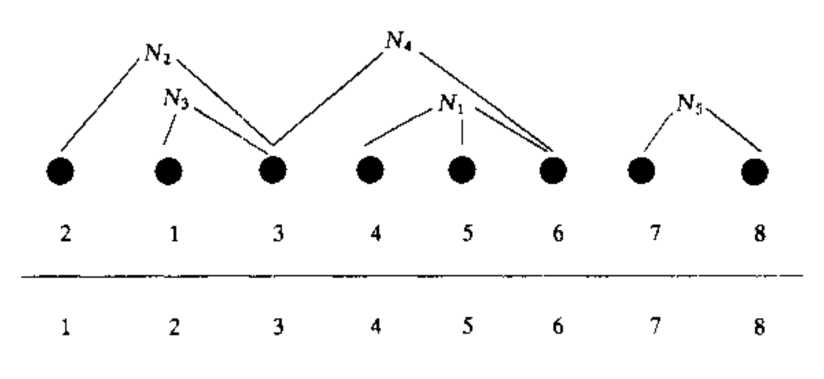


图 5-9 电路板排列

设 x 表示 n 块电路板的一个排列,即在机箱的第 i 个插槽中插入电路板 x[i]。x 所确定的电路板排列密度 density(x) 定义为跨越相邻电路板插槽的最大连线数。

例如,图 5-9 中电路板排列的密度为 2,跨越插槽 2 和 3,插槽 4 和 5 以及插槽 5 和 6 的连线数均为 2。插槽 6 和 7 之间无跨越连线。其余相邻插槽之间都只有 1 条跨越连线。

在设计机箱时,插槽一侧的布线间隙由电路板排列的密度所确定。因此电路板排列问题要求对于给定电路板连接条件(连接块),确定电路板的最佳排列,使其具有最小密度。

2. 算法设计

电路板排列问题是一个 NP 难问题,因此不大可能找到解此问题的多项式时间算法。下面我们讨论用回溯法解电路板排列问题。通过系统地搜索问题解空间的排列树,找出电路板最佳排列。

算法中用整型二维数组 B 表示输入。B[i][j] 的值为 1 当且仅当电路板 i 在连接块 N_j 中。设 total[j] 是连接块 N_j 中的电路板数。对于电路板的部分排列 x[1:i],设 now[j] 是 x[1:i] 中所包含的 N_j 中的电路板数。由此可知,连接块 N_j 的连线跨越插槽 i 和 $i+1$ 当且仅当now[j] > 0 且 now[j] $\neq$ total[j]。我们可以利用这个条件来计算插槽 i 和插槽 $i+1$ 间的连线密度。

我们用类 Board 来实现电路板排列问题的回溯算法。其私有成员函数 Backtrack 完成对解空间的搜索。函数 Arrangement 调用 Backtrack 计算并返回电路板最佳排列的密度,bestx 返回电路板的最佳排列。

函数 Arrangement 创建类 Board 的一个成员 X,并初始化其相应的变量。将 now[1:n] 初始化为 0;total[j] 初始化为连接块 N_j 中所含电路板数。函数调用 X.Backtrack(1,0) 搜索排列树寻求最佳排列。在一般情况下,X.Backtrack(i,cd) 寻求最佳部分排列 x[1:$i-1$],其部分排列密度为 cd。

在算法 Backtrack 中,当 $i = n$ 时,所有 n 块电路板都已排定,其密度为 cd。由于算法仅完成那些比当前最优解更好的排列,故 cd 肯定优于 bestd。此时应更新 bestd。

当 $i < n$ 时,电路板排列尚未完成。x[1:$i-1$] 是当前扩展结点所相应的部分排列,cd 是相应的部分排列密度。在当前部分排列之后加入一块未排定的电路板,扩展当前部分排列产生当前扩展结点的一个儿子结点。对于这个儿子结点,计算新的部分排列密度 ld。仅当 ld < bestd 时,算法搜索相应的子树,否则该子树被剪去。

按上述回溯搜索策略设计的解电路板排列问题的算法可描述如下:

```
class Board {
  friend Arrangement(int * * , int, int, int []);
  private:
    void Backtrack(int i, int cd);
    int  n,        // 电路板数
         m,        // 连接块数
         * x,      // 当前解
         * bestx,  // 当前最优解
         bestd,    // 当前最优密度
         * total,  // total[j] = 连接块 j 的电路板数
         * now,    // now[j] = 当前解中所含
                   // 连接块 j 的电路板数
         * * B;    // 连接块数组
};
```

```
void Board::Backtrack(int i, int cd)
```

```
{// 回溯搜索排列树
  if (i == n) {
    for (int j = 1; j <= n; j++)
      bestx[j] = x[j];
    bestd = cd;}
  else
    for (int j = i; j <= n; j++) {
      // 选择 x[j] 为下一块电路板
      int ld = 0;
      for (int k = 1; k <= m; k++) {
        now[k] += B[x[j]][k];
        if (now[k] > 0 && total[k] != now[k])
          ld++;
        }
      // 更新 ld
      if (cd > ld) ld = cd;
      if (ld < bestd) {// 搜索子树
          Swap(x[i], x[j]);
          Backtrack(i + 1, ld);
          Swap(x[i], x[j]);}
        // 恢复状态
        for (int k = 1; k <= m; k++)
          now[k] -= B[x[j]][k];
        }
}
```

```
int Arrangement(int **B, int n, int m, int bestx[])
{
  Board X;
  // 初始化 X
  X.x = new int [n + 1];
  X.total = new int [m + 1];
  X.now = new int [m + 1];
  X.B = B;
  X.n = n;
  X.m = m;
  X.bestx = bestx;
  X.bestd = m + 1;
  // 初始化 total 和 now
  for (int i = 1; i <= m; i++) {
    X.total[i] = 0;
    X.now[i] = 0;
    }
```

```
    // 初始化 x 为单位排列并计算 total
    for (int i = 1; i <= n; i++) {
      X.x[i] = i;
      for (int j = 1; j <= m; j++)
        X.total[j] += B[i][j];
      }
    // 回溯搜索
    X.Backtrack(1,0);
    delete [] X.x;
    delete [] X.total;
    delete [] X.now;
    return X.bestd;
  }
  ……
```

3. 算法效率

在电路板排列问题解空间的排列树的每个结点处，函数 Backtrack 花费 $O(m)$ 计算时间为每个儿子结点计算密度。因此计算密度所耗费的总计算时间为 $O(mn!)$。另外，生成排列树需 $O(n!)$ 时间。更新当前最优解需 $O(mn)$ 时间。这是因为每次更新当前最优解至少使 bestd 减少 1，而算法运行结束时 bestd ≥ 0。因此最优解被更新的次数为 $O(m)$。

综上可知，解电路板排列问题的回溯算法 Backtrack 所需的计算时间为 $O(mn!)$。

5.12 连续邮资问题

1. 问题描述

假设某国家发行了 n 种不同面值的邮票，并且规定每张信封上最多只允许贴 m 张邮票。连续邮资问题要求对于给定的 n 和 m 的值，给出邮票面值的最佳设计，使得可在 1 张信封上贴出从邮资 1 开始，增量为 1 的最大连续邮资区间。例如，当 $n=5$ 和 $m=4$ 时，面值为(1,3,11,15,32)的 5 种邮票可以贴出邮资的最大连续邮资区间是 1 到 70。

2. 算法设计

对于连续邮资问题，我们用 n 元组 x[1:n] 表示 n 种不同的邮票面值，并约定它们从小到大排列。x[1] = 1 是惟一的选择。此时的最大连续邮资区间是[1:m]。接下来，x[2] 的可取值范围是[2:$m+1$]。在一般情况下，已选定 x[1:$i-1$]，此时的最大连续邮资区间是[1:r]，则接下来 x[i] 的可取值范围是[x[$i-1$] + 1:$r+1$]。由此可以看出，在用回溯法解连续邮资问题时，可以用一棵树来表示其解空间。该解空间树中各结点的度随 x 的不同取值而变化。

在下面的回溯法描述中，递归函数 Backtrack 实现对整个解空间的回溯搜索。递归函数 Backtrack 是类 Stamp 的成员。类 Stamp 的其他成员记录解空间中结点信息，以减少传给函数 Backtrack 的参数。maxvalue 记录当前已找到的最大连续邮资区间，bestx 是相应的当前最优解。数组 y 用于记录当前已选定的邮票面值 x[1:i] 能贴出各种邮资所需的最少邮票张数。换句话

说，y[k]是用不超过 m 张面值为 x[1:i]的邮票贴出邮资 k 所需的最少邮票张数。

在函数 Backtrack 中，当 $i > n$ 时，表示算法已搜索至一个叶结点，得到一个新的邮票面值设计方案 x[1:n]。如果该方案能贴出的最大连续邮资区间大于当前已找到的最大连续邮资区间 maxvalue，则更新当前最优值 maxvalue 和相应的最优解 bestx。

当 $i \leqslant n$ 时，当前扩展结点 Z 是解空间中的一个内部结点。在该结点处 x[1:$i-1$]能贴出的最大连续邮资区间为 $r-1$。因此，在结点 Z 处，x[i]的可取值范围是[x[$i-1$] + 1:r]，从而，结点 Z 有 r − x[$i-1$]个儿子结点。算法对当前扩展结点 Z 的每一个儿子结点，以深度优先的方式递归地对相应子树进行搜索。

解连续邮资问题的回溯算法可描述如下：

```
class Stamp {
   friend int MaxStamp(int, int, int []);
   private:
     void Backtrack(int i,int r);
     int   n,          // 邮票面值数
           m,          // 每张信封最多允许贴的邮票数
           maxvalue,   // 当前最优值
           maxint,     // 大整数
           maxl,       // 邮资上界
           * x,        // 当前解
           * y,        // 贴出各种邮资所需最少邮票数
           * bestx;    // 当前最优解
};

void Stamp::Backtrack(int i,int r)
  {
    for (int j = 0; j <= x[i-2] * (m-1);j++)
      if (y[j] < m)
         for (int k = 1;k <= my[j];k++)
           if (y[j] + k < y[j + x[i-1]*k]) y[j + x[i-1]*k] = y[j] + k;
    while (y[r] < maxint) r++;
    if (i > n) {
      if (r - 1 > maxvalue) {
        maxvalue = r - 1;
        for (int j = 1; j <= n; j++)
          bestx[j] = x[j];
        }
      return;
    }
    int *z = new int [maxl + 1];
    for (int k = 1;k <= maxl;k++)
      z[k] = y[k];
    for (int j = x[i-1] + 1;j <= r;j++) {
```

```
                x[i] = j;
                Backtrack(i + 1,r);
                for (int k = 1;k < = maxl;k ++)
                    y[k] = z[k];}
        delete [] z;
    }
```

```
int MaxStamp(int n, int m, int bestx[])
{
    Stamp X;
    int maxint = 32767;
    int maxl = 1500;
    X.n = n;
    X.m = m;
    X.maxvalue = 0;
    X.maxint = maxint;
    X.maxl = maxl;
    X.bestx = bestx;
    X.x = new int [n + 1];
    X.y = new int [maxl + 1];
    for (int i = 0; i < = n; i ++) X.x[i] = 0;
    for (int i = 1; i < = maxl; i ++) X.y[i] = maxint;
    X.x[1] = 1;
    X.y[0] = 0;
    X.Backtrack(2,1);
    delete [] X.x;
    delete [] X.y;
    return X.maxvalue;
}
```

5.13 回溯法的效率分析

通过前面的具体实例的讨论容易看出，一个回溯算法的效率在很大程度上依赖于以下几个因素：

(1) 产生 x[k] 的时间；

(2) 满足显约束的 x[k] 值的个数；

(3) 计算约束函数 Constraint 的时间；

(4) 计算上界函数 Bound 的时间；

(5) 满足约束函数和上界函数约束的所有 x[k] 的个数。

一般地说，一个好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算

量较大。因此，在选择约束函数时通常存在着生成结点数与约束函数计算量之间的折衷。我们希望总的计算时间较少，而不只考虑生成的结点数少或约束函数容易计算。

为了提高效率，通常可以应用所谓的"重排原理"。对于许多问题而言，在进行搜索试探时选取 x[i] 值的顺序是任意的。这就提示我们，在其他条件相当的前提下，让可取值最少的 x[i] 优先将更为有效。从图 5-10 所示的同一问题的两棵不同的解空间树，可以体会到这种策略的效力。

在图 5-10 (a) 中，若从第 1 层剪去 1 棵子树，则从所有应当考虑的 3 元组中一次消去 12 个 3 元组。对于图 5-10 (b)，虽然同样是从第 1 层剪去 1 棵子树，却只从应当考虑的 3 元组中消去 8 个 3 元组。前者的效果明显比后者好。

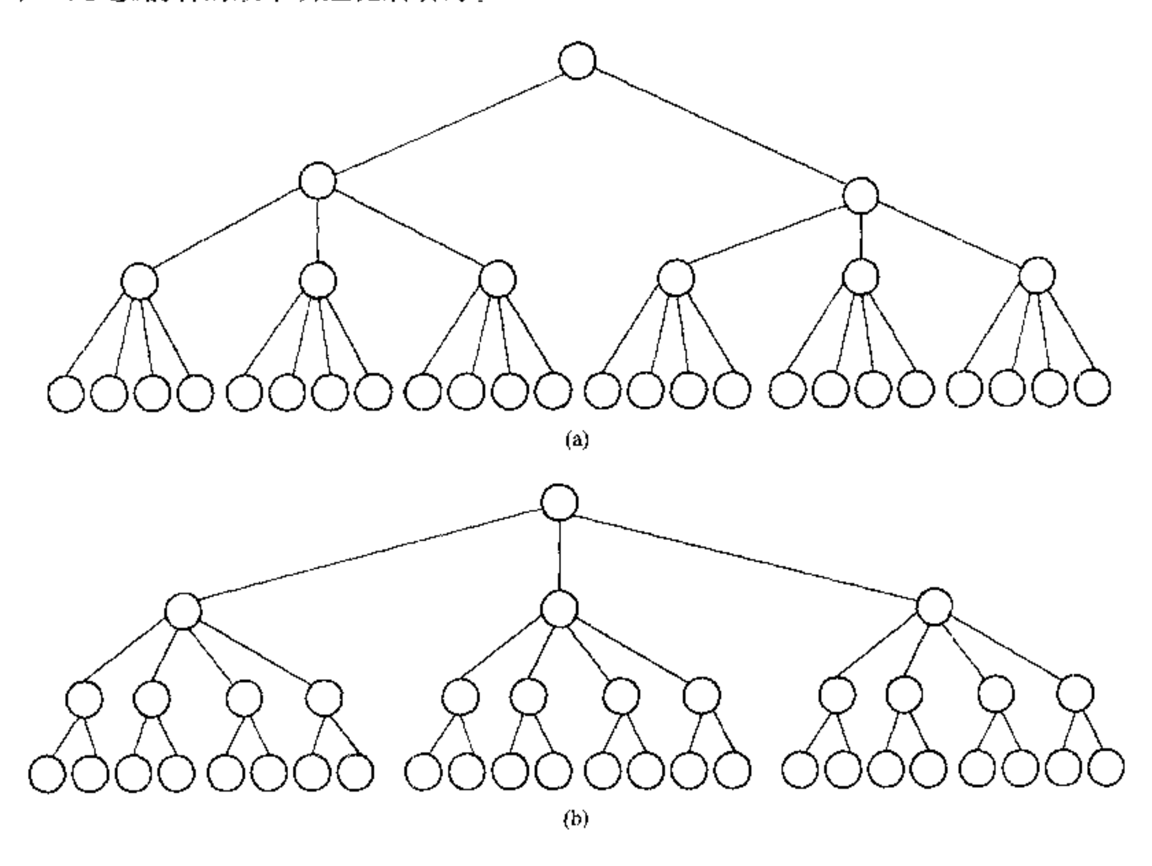


图 5-10　同一问题的 2 棵不同的解空间树

解空间的结构一经选定，影响回溯法效率的前三个因素就可以确定，只剩下生成结点的数目是可变的，它将随问题的具体内容以及结点的不同生成方式而变动。即使是对同一问题的不同实例，回溯法所产生的结点数也会有很大变化。对于一个实例，回溯法可能只产生 $O(n)$ 个结点。而对另一个非常相近的实例，回溯法可能就会产生解空间中所有结点。如果解空间的结点数是 2^n 或 $n!$，则在最坏情况下，回溯法的时间耗费一般为 $O(p(n)2^n)$ 或 $O(q(n)n!)$。其中，$p(n)$ 和 $q(n)$ 均为 n 的多项式。对一个具体问题来说，回溯法的有效性往往就体现在当问题实例的规模 n 较大时，它能够用很少的时间就求出问题的解。而对于一个问题的具体实例，我们又很难预测回溯法的算法行为。特别是我们很难估计出回溯法在解这一具体实例时所产生的结点数。这是我们在分析回溯法效率时遇到的主要困难。下面我们介绍一个概率方法，用于克服这一困难。

当应用回溯法解某一具体问题的具体实例 I 时，可用蒙特卡罗方法来估算回溯法将要产

生的结点数目。该方法的主要思想是在解空间树上产生一条随机的路径，然后沿此路径来估算解空间树中满足约束条件的结点总数 m。设 x 是所产生的随机路径上的一个结点，且位于解空间树的第 i 层上。对于 x 的所有儿子结点，用约束函数检测出满足约束条件的结点数目 m_i。路径上的下一个结点是从 x 的 m_i 个满足约束函数的儿子结点中随机选取的。这条路径一直延伸到一个叶结点或者一个所有儿子结点都不满足约束条件的结点为止。通过这些 m_i 的值，就可估算出解空间树中满足约束条件的结点总数 m。在用回溯法求问题的所有解时，这个数特别有用。因为在这种情况下，解空间中所有满足约束条件的结点都必须生成。若只要求用回溯法找出问题的一个解，则所生成的结点数一般只是 m 个满足约束条件的结点中的一小部分。此时，用 m 来估计回溯法生成的结点数就过于保守。

为了从 m_i 的值求出 m 的值，还需要对约束函数做一些假定。在估计 m 时，假定所有约束函数是静态的。也就是说，在回溯法执行过程中，约束函数并不随着算法所获得信息的多少而动态地改变。进一步还假设对解空间树中同一层的结点所用的约束函数是相同的。对于大多数的回溯法，这种假定都太强了。实际上，在大多数的回溯法中，约束函数是随着搜索过程的深入而逐渐加强的。在这种情形下，按照我们所做的假定来估计 m 就显得保守。如果将约束函数的变化也加以考虑，所得出的满足约束条件的结点总数就要比我们所估计的 m 少，而且也更精确。

在静态约束函数的假设下，我们看到在第1层共有 m_0 个满足约束条件的结点。若解空间树的同一层结点具有相同的出度，则第1层上每个结点平均有 m_1 个儿子结点满足约束条件。因此，第2层有 m_0m_1 个满足约束条件的结点。同理，第3层上满足约束条件的结点个数为 $m_0m_1m_2$。依此类推，可知第 $i+1$ 层上满足约束条件的结点个数为 $m_0m_1m_2\cdots m_i$。因此，对于给定的输入 I，如果随机地产生解空间树上的一条路径，并求出 $m_0, m_1, m_2, \cdots m_i, \cdots$，则可以估计出回溯法要生成的满足约束条件的结点总数 m 为：$1 + m_0 + m_0m_1 + m_0m_1m_2 + \cdots$。

下面的算法 Estimate 依据上述思想来计算回溯法生成的结点总数 m。该算法从解空间树的根结点开始选取一条随机路径。其中函数调用 Size(T) 得到的是集合 T 的大小；Choose(T) 则是从集合 T 中随机地选取一个元素。

```
int Estimate(int n, Type * x)
{
   int m = 1, r = 1, k = 1;
   while (k <= n) {
      SetType T = x[k] 的满足约束的可取值集合;
      if (Size(T) == 0) return m;
      r *= Size(T);
      m += r;
      x[k] = Choose(T);
      k ++;}
   return m;
}
```

当用回溯法求解某一具体问题时，可用算法 Estimate 估算回溯法生成的结点数。若要估计

得更精确些，可选取若干条不同的随机路径（通常不超过 20 条），分别对各随机路径估计结点总数，然后再取这些结点总数的平均值，得到 m 的估算值

例如，对于 8 后问题，要在 8×8 的棋盘中放进 8 个皇后，其放法的组合数是很大的。利用显约束排除那些有 2 个皇后在同一行或同一列的放法，也还有 8!种不同的放法。我们可以用算法 Estimate 来估计解 n 后问题的回溯法 nQueen 所产生的结点总数。容易看出，对于该问题，约束函数的静态假设是成立的，即在算法的搜索过程中，约束函数并没有改变。另外，在解空间树中，同一层的所有结点都有相同的出度。图 5-11 给出了算法 Estimate 产生的 5 条随机路径所相应的 8×8 棋盘状态。当需要在棋盘上某行放入一个皇后时，所放的列是随机地选取的。它与已在棋盘上的其他后互不攻击。

在图中棋盘下面列出了每一层的结点可能生成的满足约束条件的结点数，即 $m_0,m_1,m_2,\cdots m_i,\cdots$，以及由此随机路径估算出的结点总数 m 的值。由这 5 条随机路径可以得到 m 的平均值为 1702。而 8 后问题的解空间树的结点总数是

$$1+\sum_{j=0}^{7}\left(\prod_{i=0}^{j}(8-i)\right)=109\ 601$$

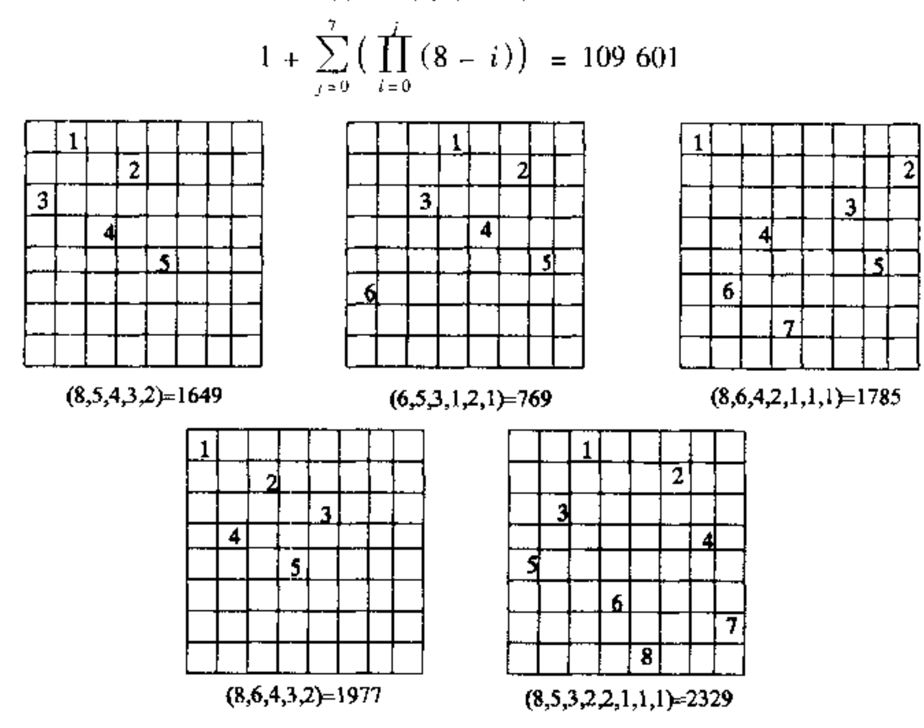


图 5-11　解空间树中 5 条随机路径所对应的棋盘状态

因此，回溯法产生的结点数 m 是解空间树的结点总数的 1.55% 左右。这说明回溯法的效率大大高于穷举法。

习题 5

5-1　用教材中提到的改进策略 1 重写装载问题的回溯法，使改进后算法的计算时间复杂性为 $O(2^n)$。

5-2　用教材中提到的改进策略 2 重写装载问题的回溯法，使改进后算法的计算时间复杂性为 $O(2^n)$。

5-3　试设计一个解子集和问题的递归回溯法。注意对于子集和问题，一旦找到和为 c 的

子集，算法即可终止。算法中不必记录当前最优解，也不必用数组 x 记录当前路径。问题的解可在找到和为 c 的子集后重新构造。

5-4　重写 0-1 背包问题的回溯法，使算法运行结束后能输出最优解。

5-5　试设计一个解最大团问题的迭代回溯法

5-6　试设计一个解最大独立集问题的回溯法。

5-7　设 G 是一个有 n 个顶点的有向图，从顶点 i 发出的边的最大费用记为 $\max(i)$。

(1) 证明任何一个旅行售货员回路的费用都不超过 $\sum_{i=1}^{n}\max(i)+1$。

(2) 在解旅行售货员问题的回溯法中，用上面的界作为 bestc 的初始值，重写该算法，并尽可能地简化代码。

5-8　设 G 是一个有 n 个顶点的有向图，从顶点 i 发出的边的最小费用记为 $\min(i)$。

(1) 证明图 G 的所有前缀为 $x[1:i]$ 的旅行售货员回路的费用至少为 $\sum_{j=2}^{i}a(x_{j-1},x_j)+\sum_{j=i}^{n}\min(x_j)$，其中，$a(u,v)$ 是边 (u,v) 的费用。

(2) 利用上述结论设计一个高效的上界函数，重写旅行售货员问题的回溯法，并与教材中的算法进行比较。

5-9　最小长度电路板排列问题。在电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如，在图 5-9 所示的电路板排列中，连接块 N_4 的第 1 块电路板在插槽 3 中，它的最后 1 块电路板在插槽 6 中，因此 N_4 的长度为 3。同理 N_2 的长度为 2。图 5-9 中连接块最大长度为 3。试设计一个回溯法找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

5-10　最小重量机器设计问题。设某一机器由 n 个部件组成，每一种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。试设计一个算法，给出总价格不超过 c 的最小重量机器设计。

5-11　试设计一个用回溯法搜索子集空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解装载问题和 0-1 背包问题。

5-12　用排列空间树做习题 5-11。

5-13　试设计一个用回溯法搜索一般解空间的函数。该函数的参数包括：生成解空间中下一扩展结点的函数、结点可行性判定函数和上界函数等必要的函数，并将此函数用于解装载问题和 0-1 背包问题。

5-14　运动员最佳配对问题。一个羽毛球队有男女运动员各 n 人，给定 2 个 $n\times n$ 矩阵 $\boldsymbol{P}$ 和 $\boldsymbol{Q}$。P[i][j] 是男运动员 i 和女运动员 j 配对组成混合双打时的竞赛优势；Q[i][j] 则是女运动员 i 和男运动员 j 配合时的竞赛优势。显然，由于技术的配合和心理状态等各种因素的影响，P[i][j] 不一定等于 Q[j][i]。设计一个算法，计算出男女运动员的最佳配对法，使各组男女双方竞赛优势乘积的总和达到最大。

5-15　无分隔符字典问题。设 $\sum=\{a_1,a_2,\cdots,a_n\}$ 是 n 个互不相同的符号组成的符号集。$L_k=\{\beta_1\beta_2\cdots\beta_k \mid \beta_i\in\sum, 1\leqslant i\leqslant k\}$ 是 $\sum$ 中字符组成的长度为 k 的字符串全体。$S\subseteq L_k$ 是 L_k 的一个无分隔符字典是指对任意 $a_1a_2\cdots a_k\in S$ 和 $b_1b_2\cdots b_k\in S$，则

$$\{a_2a_3\cdots a_kb_1, a_3a_4\cdots b_1b_2, \cdots, a_kb_1b_2\cdots b_{k-1}\} \cap S = \varnothing$$

无分隔符字典问题要求对给定的 n, $\sum$ 以及正整数 k,编程计算 L_k 的最大无分隔符字典。

5-16 无和集问题。设 S 是一个正整数集合。S 是一个无和集当且仅当 $x, y \in S$ 蕴含 $x + y \notin S$。对于任意正整数 k,如果可将 $\{1,2,\cdots,k\}$ 划分为 n 个无和子集 $S_1, S_2, \cdots, S_n$,则称正整数 k 是 n 可分的。记 $F(n) = \max\{k \mid k$ 是 n 可分的$\}$。试设计一个算法,对任意给定的 n,计算 $F(n)$ 的值。

5-17 四色方柱问题(Instant Insanity)。设有 4 个立方体,每个立方体的每一面用红、黄、蓝、绿 4 种颜色之一染成。我们要把这 4 个立方体叠成一个方形柱体,使得柱体的 4 个侧面的每一侧均有 4 种不同的颜色。同时,4 个顶面和 4 个底面也都有 4 种不同的颜色。试设计一个回溯算法,计算出 4 个立方体的一种满足要求的叠置方案。

5-18 整数变换问题。关于整数 i 的变换 f 和 g 定义如下:$f(i) = 3i$;$g(i) = \lfloor i/2 \rfloor$。试设计一个算法,对于给定的两个整数 n 和 m,用最少的 f 和 g 变换次数将 n 变换为 m。例如,我们可以将整数15用4次变换将它变换为整数4: $4 = gfgg(15)$。当整数 n 不可能变换为整数 m 时,算法应如何处理?

5-19 排列宝石问题。设有 n 种不同的颜色,同一种形状的 n 颗宝石分别具有这 n 种不同的颜色。现有 n 种不同形状的宝石共 n^2 颗,欲将这 n^2 颗宝石排列成 n 行 n 列的一个方阵,使方阵中每一行和每一列的宝石都有 n 种不同形状和 n 种不同颜色。试设计一个算法计算出对于给定的 n 有多少种不同的宝石排列方案。

5-20 网络设计问题。石油传输网络通常可表示为一个非循环带权有向图 G。G 中有一个称为源的顶点 s。石油从该顶点输送至 G 中其他顶点。图 G 中每条边的权表示该边连接的两个顶点间的距离。网络中的油压随距离增大而减小。为了保证整个输油网络正常工作,需要维持网络中的最低油压 $P_{\min}$。为此需要在网络的某些或全部顶点处设置增压器。在设置增压器的顶点处油压可升至最大值 $P_{\max}$。油压从 $P_{\max}$ 减至 $P_{\min}$ 可使石油传输的距离至少为 d。试设计一个算法,计算出网络中增压器的最优放置方案,使得用最少的增压器保证石油运输畅通。

5-21 罗密欧与朱丽叶的迷宫。罗密欧与朱丽叶身处一个 $m \times n$ 的迷宫中,如图 5-12 所示。每一个方格表示迷宫中的一个房间。这 $m \times n$ 个房间中有一些房间是封闭的,不允许任何人进入。在迷宫中任何位置均可沿 8 个方向进入未封闭的房间。罗密欧位于迷宫的(p,q)方格中,他必须找出一条通向朱丽叶所在的(r,s)方格的路。在抵达朱丽叶方格之前,他必须走遍所有未封闭的房间各一次,而且要使到达朱丽叶方格的转弯次数为最少。每改变一次前进方向算作转弯一次。请设计一个算法帮助罗密欧找出这样一条道路。

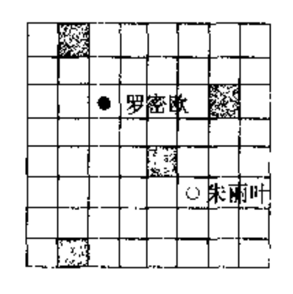


图 5-12 罗密欧与朱丽叶的迷宫

5-22 工作分配问题。设有 n 件工作要分配给 n 个人去完成。将工作 i 分配给第 j 个人所需的费用为 c_{ij}。试设计一个算法,为每一个人都分配 1 件不同的工作,并使总费用达到最小。

5-23 独立钻石跳棋问题。图 5-13 所示的 33 个方格顶点摆放着 32 枚棋子,仅中央的顶点空着未摆放棋子。下棋的规则是任一棋子可以沿水平或垂直方向跳过与其相邻的棋子进入空着的顶点并吃掉被跳过的棋子。试设计一个算法,找出一种下棋方法,使得最终棋盘上只剩下

一个棋子在棋盘中央

5-24 智力拼图问题。设有12个平面图形如图5-14所示。每个图形的形状互不相同，但它们都是由5个大小相同的正方形所组成。在图5-14中，这12个图形拼接成一个6×10的矩形。试设计一个算法，计算出有多少种不同的方案可用这12个图形拼接成一个6×10的矩形。

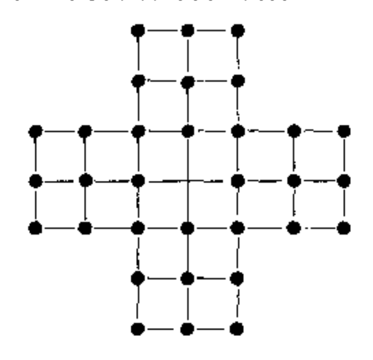


图5-13 独立钻石跳棋初始布局

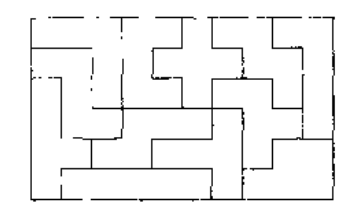


图5-14 智力拼图

5-25 布线问题。假设我们要将一组元件安装在一块线路板上，为此需要设计一个线路板布线方案。各元件的连线数由连线矩阵conn给出。元件 i 和元件 j 之间的连线数为 $\mathrm{conn}(i,j)$。如果将元件 i 安装在线路板上位置 r 处，而将元件 j 安装在线路板上位置 s 处，则元件 i 和元件 j 之间的距离为 $\mathrm{dist}(r,s)$。确定了所给的 n 个元件的安装位置，就确定了一个布线方案。与此布线方案相应的布线成本为 $\sum_{1 \leqslant i \leqslant j \leqslant n} \mathrm{conn}(i,j) * \mathrm{dist}(r,s)$。试设计一个算法找出所给 n 个元件的布线成本最小的布线方案。

5-26 最佳调度问题。假设有 n 个任务要由 k 个可并行工作的机器来完成。完成任务 i 需要的时间为 t_i。试设计一个算法找出完成这 n 个任务的最佳调度，使得完成全部任务的时间最早。

5-27 无优先级运算问题。设有 a,b,c,d,e 5个整数和运算符 +，-，*，/，且运算符无优先级。如 $2+3*5=25$。对于给定的整数 n，试设计一个算法，用以上给出的5个数和运算符，产生整数 n，且用的运算次数最少。

5-28 最大 k 乘积问题。设 I 是一个 n 位十进制整数。如果将 I 划分为 k 段，则可得到 k 个整数。这 k 个整数的乘积称为 I 的一个 k 乘积。试设计一个算法，对于给定的 I 和 k，求出 I 的最大 k 乘积。

5-29 世界名画陈列馆问题。世界名画陈列馆由 $m \times n$ 个陈列室组成。为了防止名画被盗，需要在陈列室中设置警卫机器人哨位。每个警卫机器人除了监视它所在的陈列室外，还可以监视与它所在的陈列室相邻的上、下、左、右4个陈列室。试设计一个安排警卫机器人哨位的算法，使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下，且所用的警卫机器人数最少。

5-30 世界名画陈列馆问题。在上题中，如果要求每一个陈列室仅受一个警卫机器人监视，则应如何设置警卫机器人的哨位。

5-31 魔方(Rubik's Cube)问题。给定魔方的初始状态和目标状态，试设计一个算法计算出从初始状态到目标状态所需的最少旋转次数。当初始状态不可能变换为目标状态时，算法是否会陷入死循环?

第6章　分支限界法

学习要点

· 理解分支限界法的剪枝搜索策略
· 掌握分支限界法的算法框架：
(1) 队列式(FIFO)分支限界法
(2) 优先队列式分支限界法
· 通过下面的应用范例学习分支限界法的设计策略：
(1) 单源最短路径问题
(2) 装载问题
(3) 布线问题
(4) 0-1背包问题
(5) 最大团问题
(6) 旅行售货员问题
(7) 电路板排列问题
(8) 批处理作业调度问题

分支限界法类似于回溯法，也是一种在问题的解空间树 T 上搜索问题解的算法。但在一般情况下，分支限界法与回溯法的求解目标不同。回溯法的求解目标是找出 T 中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

由于求解目标不同，导致分支限界法与回溯法在解空间树 T 上的搜索方式也不相同。回溯法以深度优先的方式搜索解空间树 T，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树 T。分支限界法的搜索策略是，在扩展结点处，先生成其所有的儿子结点(分支)，然后再从当前的活结点表中选择下一个扩展结点。为了有效地选择下一扩展结点，以加速搜索的进程，在每一活结点处，计算一个函数值(限界)，并根据这些已计算出的函数值，从当前活结点表中选择一个最有利的结点作为扩展结点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解。这种方法就称为分支限界法。人们已经用分支限界法解决了大量离散最优化的实际问题。

6.1　分支限界法的基本思想

分支限界法常以广度优先或以最小耗费(最大效益)优先的方式搜索问题的解空间树。问题的解空间树是表示问题解空间的一棵有序树，常见的有子集树和排列树。在搜索问题的解空间树时，分支限界法与回溯法对当前扩展结点所采用的扩展方式不同。在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，那些导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点

被加入活结点表中。此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。

从活结点表中选择下一扩展结点的不同方式导致不同的分支限界法。最常见的有以下两种方式：

(1) 队列式(FIFO)分支限界法

队列式分支限界法将活结点表组织成一个队列，并按队列的先进先出原则选取下一个结点为当前扩展结点。

(2) 优先队列式分支限界法

优先队列式的分支限界法将活结点表组织成一个优先队列，并按优先队列中规定的结点优先级选取优先级最高的下一个结点成为当前扩展结点。

优先队列中规定的结点优先级常用一个与该结点相关的数值 p 来表示。结点优先级的高低与 p 值的大小相关。最大优先队列规定 p 值较大的结点优先级较高。在算法实现时通常用一个最大堆来实现最大优先队列，用最大堆的 Deletemax 运算抽取堆中下一个结点成为当前扩展结点，体现最大效益优先的原则。类似地，最小优先队列规定 p 值较小的结点优先级较高。在算法实现时通常用一个最小堆来实现最小优先队列，用最小堆的 Deletemin 运算抽取堆中下一个结点成为当前扩展结点，体现最小费用优先的原则。

用优先队列式分支限界法解具体问题时，应根据具体问题的特点确定选用最大优先队列或最小优先队列来表示解空间的活结点表。

例如，考虑 $n = 3$ 时0-1 背包问题的一个实例如下：$w = [16,15,15]$，$p = [45,25,25]$，$c = 30$。这个例子我们在第 5 章中曾经讨论过，其解空间是图 5-1 中的子集树。

用队列式分支限界法解此问题时，用一个队列来存储活结点表。算法从根结点 A 开始。初始时活结点队列为空，结点 A 是当前扩展结点。结点 A 的 2 个儿子结点 A 和 B 均为可行结点，故将这 2 个儿子结点按从左到右的顺序加入活结点队列，并且舍弃当前扩展结点 A。依先进先出原则，下一个扩展结点是活结点队列的队首结点 B。扩展结点 B 得到其儿子结点 D 和 E。由于 D 是不可行结点，故被舍去。E 是可行结点，被加入活结点队列。接下来，C 成为当前扩展结点，它的 2 个儿子结点 F 和 G 均为可行结点，因此被加入到活结点队列中。扩展下一个结点 E 得到结点 J 和 K。J 是不可行结点，因而被舍去。K 是一个可行的叶结点，表示所求问题的一个可行解，其价值为 45。

当前活结点队列的队首结点 F 成为下一个扩展结点。它的 2 个儿子结点 L 和 M 均为叶结点。L 表示获得价值为 50 的可行解；M 表示获得价值为 25 的可行解。G 是最后的一个扩展结点，其儿子结点 N 和 O 均为可行叶结点。最后，活结点队列已空，算法终止。算法搜索得到最优值为 50。

从这个例子容易看出，队列式分支限界法搜索解空间树的方式与解空间树的广度优先遍历算法极为相似。惟一的不同之处是队列式分支限界法不搜索以不可行结点为根的子树。

优先队列式分支限界法也是从根结点 A 开始搜索解空间树的。我们用一个极大堆来表示活结点表的优先队列，该优先队列的优先级定义为活结点所获得的价值。初始时堆为空，扩展结点 A 得到它的 2 个儿子结点 B 和 C。这 2 个结点均为可行结点，因此被加入到堆中，结点 A 被舍弃。结点 B 获得的当前价值是 40，而结点 C 的当前价值为 0。由于结点 B 的价值大于结点 C 的价值，所以结点 B 是堆中最大元素，从而成为下一个扩展结点。扩展结点 B 得到结点 D 和 E。D 不是可行结点，因而被舍去。E 是可行结点被加入到堆中。E 的价值为 40，成为当前堆中

最大元素，从而成为下一个扩展结点。扩展结点 E 得到2个叶结点 J 和 K。J 是不可行结点被舍弃。K 是一个可行叶结点，表示所求问题的一个可行解，其价值为45。此时，堆中仅剩下一个活结点 C，它成为当前扩展结点。它的2个儿子结点 F 和 G 均为可行结点，因此被插入到当前堆中。结点 F 的价值为25，是堆中最大元素，成为下一个扩展结点。结点 F 的2个儿子结点 L 和 M 均为叶结点。叶结点 L 相应于价值为50的可行解。叶结点 M 相应于价值为25的可行解。叶结点 L 所相应的解成为当前最优解。最后，结点 G 成为扩展结点，其儿子结点 N 和 O 均为叶结点，它们的价值分别为25和0。接下来，存储活结点的堆已空，算法终止。算法搜索得到最优值为50，相应的最优解是从根结点 A 到结点 L 的路径(0,1,1)。

当我们要寻求问题的一个最优解时，与我们在讨论回溯法时类似地可以用剪枝函数来加速搜索。该函数给出每一个可行结点相应的子树可能获得的最大价值的上界。如果这个上界不会比当前最优值更大，则说明相应的子树中不含问题的最优解，因而可以剪去。另一方面，我们也可以将上界函数确定的每个结点的上界值作为优先级，以该优先级的非增序抽取当前扩展结点。这种策略有时可以更迅速地找到最优解。

我们再来看一个4城市旅行售货员的例子，如图5-3所示，该问题的解空间树是一棵排列树。解此问题的队列式分支限界法以排列树中结点 B 作为初始扩展结点。此时，活结点队列为空。由于从图 G 的顶点1到顶点2，3和4均有边相连，所以结点 B 的儿子结点 C，D，E 均为可行结点，它们被加入到活结点队列中，并舍去当前扩展结点 B。当前活结点队列中的队首结点 C 成为下一个扩展结点。由于图 G 的顶点2到顶点3和4有边相连，故结点 C 的2个儿子结点 F 和 G 均为可行结点，从而被加入到活结点队列中。接下来，结点 D 和结点 E 相继成为扩展结点而被扩展。此时，活结点队列中的结点依次为 F，G，H，I，J，K。

结点 F 成为下一个扩展结点，其儿子结点 L 是一个叶结点。我们找到了一条旅行售货员回路，其费用为59。从下一个扩展结点 G 得到叶结点 M，它相应的旅行售货员回路的费用为66。结点 H 依次成为扩展结点，得到结点 N 相应的旅行售货员回路，其费用为25。这是当前最好的一条回路。下一个扩展结点是结点 I，由于从根结点到叶结点 I 的费用26已超过了当前最优值，故没有必要扩展结点 I，以结点 I 为根的子树被剪去。最后，结点 J 和 K 被依次扩展，活结点队列成为空，算法终止。算法搜索得到最优值为25，相应的最优解是从根结点到结点 N 的路径(1,3,2,4,1)。

解同一问题的优先队列式分支限界法用一极小堆来存储活结点表。其优先级是结点的当前费用。算法还是从排列树的结点 B 和空优先队列开始。结点 B 被扩展后，它的3个儿子结点 C，D 和 E 被依次插入堆中。此时，由于 E 是堆中具有最小当前费用(4)的结点，所以处于堆顶的位置，它自然成为下一个扩展结点。结点 E 被扩展后，其儿子结点 J 和 K 被插入当前堆中，它们的费用分别为14和24。此时，堆顶元素是结点 D，它成为下一个扩展结点。它的2个儿子结点 H 和 I 被插入堆中。此时堆中含有结点 C，H，I，J，K。在这些结点中，结点 H 具有最小费用，从而它成为下一个扩展结点。扩展结点 H 后得到一条旅行售货员回路(1,3,2,4,1)，相应的费用为25。接下来，结点 J 成为扩展结点，由此得到另一条费用为25的回路(1,4,2,3,1)。此后的2个扩展结点是结点 K 和 I。由结点 K 得到的可行解费用高于当前最优解，结点 I 本身的费用已高于当前最优解，从而它们都不能得到更好的解。最后，优先队列为空，算法终止

与0-1背包问题的例子类似，可以用一个限界函数在搜索过程中裁剪子树，以减少产生的活结点。此时剪枝函数是当前结点扩展后得到的最小费用的一个下界。如果在当前扩展结点处，这个下界不比当前最优值更小，则以该结点为根的子树可以被剪去。另一方面，我们也可以

把每个结点的下界作为优先级，依非减序从活结点优先队列中抽取下一个扩展结点。

6.2 单源最短路径问题

单源最短路径问题适合于用分支限界法求解。我们先来看单源最短路径问题的一个实例。在图6-1所给的有向图 G 中，每一边都有一个非负边权。我们要求图 G 的从源顶点 s 到目标顶点 t 之间的最短路径。解单源最短路径问题的优先队列式分支限界法用一极小堆来存储活结点表。其优先级是结点所对应的当前路长。算法从图 G 的源顶点 s 和空优先队列开始。结点 s 被扩展后，它的3个儿子结点被依次插入堆中。此后，算法从堆中取出具有最小当前路长的结点作为当前扩展结点，并依次检查与当前扩展结点相邻的所有顶点。如果从当前扩展结点 i 到顶点 j 有边可达，且从源出发，途经顶点 i 再到顶点 j 的所相应的路径的长度小于当前最优路径长度，则将该顶点作为活结点插入到活结点优先队列中。这个结点的扩展过程一直继续到活结点优先队列为空时为止。

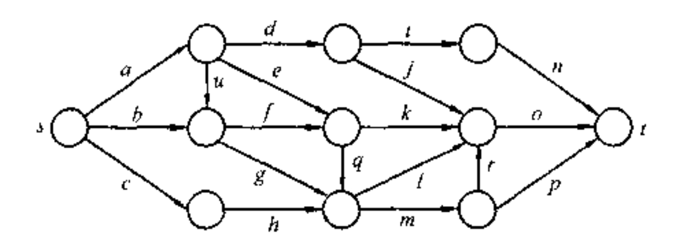


图6-1　有向图 G

图6-2是用优先队列式分支限界法解图6-1所给的有向图 G 的单源最短路径问题所产生的解空间树。其中，每一个结点旁边的数字表示该结点所对应的当前路长。由于图 G 中各边的权均非负，所以结点所对应的当前路长也是解空间树中以该结点为根的子树中所有结点所对应的路长的一个下界。在扩展结点的过程中，一旦发现一个结点的下界不小于当前找到的最短路长，则剪去以该结点为根的子树。

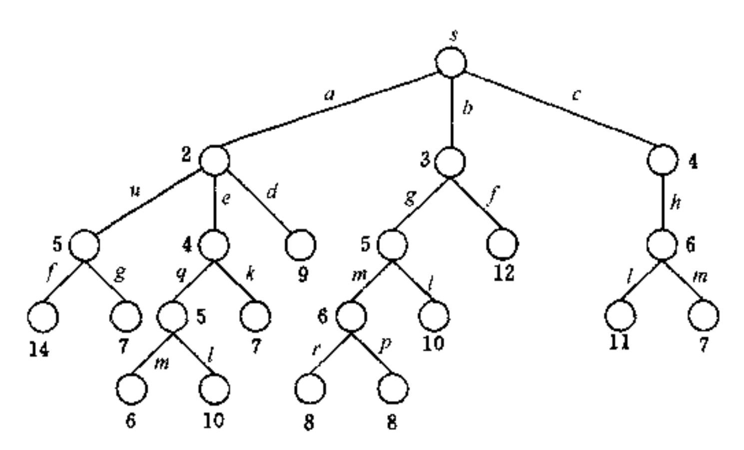


图6-2　有向图 G 的单源最短路径问题的解空间树

在算法中，我们还利用结点间的控制关系进行剪枝。例如在上例中，从源顶点 s 出发，经过边 a,e,q（路长为5）和经过边 c,h（路长为6）的2条路径到达图 G 的同一顶点。在该问题的解空间树中，这2条路径相应于解空间树的2个不同的结点 A 和 B。由于结点 A 所相应的路长小于结点 B 所相应的路长，因此以结点 A 为根的子树中所包含的从 s 到 t 的路长小于以结点 B 为

根的子树中所包含的从 s 到 t 的路长。因而可以将以结点 B 为根的子树剪去 这时称结点 A 控制了结点 B。

下面给出的算法要找出从源顶点 s 到图 G 中所有其他顶点之间的最短路径，主要利用结点控制关系进行剪枝。在一般情况下，如果解空间树中以结点 y 为根的子树中所含的解优于以结点 x 为根的子树中所含的解，则结点 y 控制了结点 x，以被控制的结点 x 为根的子树可以剪去。

在具体实现算法时，用邻接矩阵表示所给的图 G。在类 Graph 中用一个二维数组 c 存储图 G 的邻接矩阵；用数组 dist 记录从源到各顶点的距离；用数组 prev 记录从源到各顶点的路径上的前驱顶点。

由于我们要找的是从源到各顶点的最短路径，所以我们选用最小堆表示活结点优先队列。最小堆中元素的类型为 MinHeapNode。该类型结点包含域 i，用于记录该活结点所表示的图 G 中相应顶点的编号；length 表示从源到该顶点的距离。

```
template < class Type >
class Graph {
    friend void main(void);
    public:
        void ShortestPaths(int);
    private:
        int n,              // 图 G 的顶点数
            * prev;         // 前驱顶点数组
    Type * * c,             // 图 G 的邻接矩阵
            * dist;         // 最短距离数组
};
```

```
template < class Type >
class MinHeapNode {
  friend Graph < Type > ;
  public:
    operator int () const {return length;}
  private:
    int i;          // 顶点编号
    Type length;  // 当前路长
};
```

具体算法可描述如下：

```
template < class Type
void Graph < Type > : : ShortestPaths(int v)
{// 单源最短路径问题的优先队列式分支限界法
    // 定义最小堆的容量为 1000
    MinHeap <  MinHeapNode < Type >  >  H(1000);
```

```
        // 定义源为初始扩展结点
        MinHeapNode< Type>  E;
        E.i = v;
        E.length = 0;
        dist[v] = 0;
        // 搜索问题的解空间
        while (true) {
         for (int j = 1; j < = n; j+ + )
          if ((c[E.i][j] < inf) & &(E.length + c[E.i][j] < dist[j])) {
            // 顶点 i 到顶点 j 可达,且满足控制约束
            dist[j] = E.length + c[E.i][j];
            prev[j] = E.i;
            // 加入活结点优先队列
            MinHeapNode< Type>  N;
            N.i = j;
            N.length = dist[j];
            H.Insert(N);}
         try {H.DeleteMin(E);}          // 取下一扩展结点
         catch (OutOfBounds) {break;}   // 优先队列空
         }
    }
```

算法开始时创建一个容量为 1 000 的最小堆,用于表示活结点优先队列。堆中每个结点的 length 值是优先队列的优先级。接着算法将源顶点 v 初始化为当前扩展结点。

算法中 while 循环体完成对解空间内部结点的扩展。对于当前扩展结点,算法依次检查与当前扩展结点相邻的所有顶点。如果从当前扩展结点 i 到顶点 j 有边可达,且从源出发,途经顶点 i 再到顶点 j 的所相应的路径的长度小于当前最优路径长度,则将该顶点作为活结点插入到活结点优先队列中。完成对当前结点的扩展后,算法从活结点优先队列中取出下一个活结点作为当前扩展结点,重复上述结点的分支扩展。这个结点的扩展过程一直继续到活结点优先队列为空时为止。算法结束后,数组 dist 返回从源到各顶点的最短距离。相应的最短路径可利用从前驱顶点数组 prev 记录的信息构造出来。

6.3 装载问题

装载问题已在第 5 章中详细描述,其实质是要求第 1 艘船的最优装载。装载问题是一个子集选取问题,因此其解空间树是一棵子集树。

1. 队列式分支限界法

解装载问题的队列式分支限界法只求出所要求的最优值,稍后将进一步讨论求出最优解。函数 MaxLoading 具体实施对解空间的分支限界搜索。其中队列 Q 用于存放活结点表。在队列 Q 中用 weight 表示每个活结点所相应的当前载重量。当 weight = -1 时,表示队列已到达解空间树同一层结点的尾部。

函数 EnQueue 用于将活结点加入到活结点队列中。该函数首先检查 i 是否等于 n。如果 $i = n$，则表示当前活结点为一个叶结点。由于叶结点不会被进一步扩展，因此不必加入到活结点队列中。此时只要检查该叶结点表示的可行解是否优于当前最优解，并适时更新当前最优解。当 $i < n$ 时，当前活结点是一个内部结点，应加入到活结点队列中。

函数 MaxLoading 在开始时将 i 初始化为 1，bestw 初始化为 0。此时活结点队列为空。将同层结点尾部标志 -1 加入到活结点队列中，表示此时位于第 1 层结点的尾部。Ew 存储当前扩展结点所相应的重量。在 while 循环中，首先检测当前扩展结点的左儿子结点是否为可行结点。如果是则调用 EnQueue 将其加入到活结点队列中，然后将其右儿子结点加入到活结点队列中(右儿子结点一定是可行结点)。两个儿子结点都产生后，当前扩展结点被舍弃。活结点队列中的队首元素被取出作为当前扩展结点。由于队列中每一层结点之后都有一个尾部标记 -1，故在取队首元素时，活结点队列一定不空。当取出的元素是 -1 时，再判断当前队列是否为空。如果队列非空，则将尾部标记 -1 加入活结点队列，算法开始处理下一层的活结点。

```
template <class Type>
void EnQueue(Queue<Type> &Q, Type wt,
                    Type& bestw, int i, int n)
{// 将活结点加入到活结点队列 Q 中
    if (i == n) {// 可行叶结点
        if (wt > bestw) bestw = wt;}
    else Q.Add(wt); // 非叶结点
}
```

```
template <class Type>
Type MaxLoading(Type w[], Type c, int n)
{// 队列式分支限界法，返回最优载重量
    // 初始化
    Queue<Type> Q;   // 活结点队列
    Q.Add(-1);       // 同层结点尾部标志
    int i = 1;       // 当前扩展结点所处的层
    Type Ew = 0,     // 扩展结点所相应的载重量
      bestw = 0;     // 当前最优载重量
    // 搜索子集空间树
    while (true) {
      // 检查左儿子结点
      if (Ew + w[i] <= c) // x[i] = 1
          EnQueue(Q, Ew + w[i], bestw, i, n);
      // 右儿子结点总是可行的
      EnQueue(Q, Ew, bestw, i, n); // x[i] = 0
      Q.Delete(Ew);     // 取下一扩展结点
      if (Ew == -1) {// 同层结点尾部
        if (Q.IsEmpty()) return bestw;
        Q.Add(-1);    // 同层结点尾部标志
        Q.Delete(Ew); // 取下一扩展结点
```

```
            i++;}           // 进入下一层
        }
    }
```

算法 MaxLoading 的计算时间和空间复杂性均为 $O(2^n)$。

2. 算法的改进

与解装载问题的回溯法类似,可对上述算法作进一步改进。设 bestw 是当前最优解;Ew 是当前扩展结点所相应的重量;r 是剩余集装箱的重量。则当 Ew + r ≤ bestw 时,可将其右子树剪去。

算法 MaxLoading 初始时将 bestw 置为 0,直到搜索到第一个叶结点时才更新 bestw。因此在算法搜索到第一个叶结点之前,总有 bestw = 0,r > 0,故 Ew + r > bestw 总是成立。也就是说,此时右子树测试不起作用。

为了使上述右子树测试尽早生效,应提早更新 bestw。我们知道算法最终找到的最优值是所求问题的子集树中所有可行结点相应重量的最大值。而结点所相应的重量仅在搜索进入左子树时增加。因此,我们可以在算法每一次进入左子树时更新 bestw 的值。由此可对算法作进一步改进如下:

```
template <class Type>
Type MaxLoading(Type w[], Type c, int n)
{// 队列式分支限界法,返回最优载重量
    // 初始化
    Queue <Type> Q;          // 活结点队列
    Q.Add(-1);               // 同层结点尾部标志
    int i = 1;               // 当前扩展结点所处的层
    Type Ew = 0,             // 扩展结点所相应的载重量
      bestw = 0,             // 当前最优载重量
      r = 0;                 // 剩余集装箱重量
    for (int j = 2; j <= n; j++)
      r += w[i];
    // 搜索子集空间树
    while (true) {
      // 检查左儿子结点
      Type wt = Ew + w[i]; // 左儿子结点的重量
      if (wt <= c) { // 可行结点
        if (wt > bestw) bestw = wt;
        // 加入活结点队列
        if (i < n) Q.Add(wt);}
      // 检查右儿子结点
      if (Ew + r > bestw && i < n)
        Q.Add(Ew);       // 可能含最优解
      Q.Delete(Ew);      // 取下一扩展结点
      if (Ew == -1) {  // 同层结点尾部
```

```
            if (Q.IsEmpty()) return bestw;
            Q.Add(-1);        // 同层结点尾部标志
            Q.Delete(Ew);     // 取下一扩展结点
            i++;              // 进入下一层
            r -= w[i];}       // 剩余集装箱重量
        }
    ⋮
```

当算法要将一个活结点加入活结点队列时,wt 的值不会超过 bestw,故不必更新 bestw。因此算法中可直接将该活结点插入到活结点队列中,不必动用函数 EnQueue 来完成插入。

3. 构造最优解

为了在算法结束后能方便地构造出与最优值相应的最优解,算法必须存储相应子集树中从活结点到根结点的路径。为此,可在每个结点处设置指向其父结点的指针,并设置左、右儿子标志。与此相应的数据类型由 QNode 表示。

```
template < class Type >
class QNode {
  friend void EnQueue(Queue < QNode < Type > * > &, Type,
    int, int, Type, QNode < Type > *, QNode < Type > * &, int *, bool);
  friend Type MaxLoading(Type *, Type, int, int *);
  private:
    QNode *parent;      // 指向父结点的指针
    bool LChild;        // 左儿子标志
    Type weight;        // 结点所相应的载重量
};
```

将活结点加入到活结点队列中的函数 EnQueue 作相应的修改如下:

```
template < class Type >
void EnQueue(Queue < QNode < Type > * > &Q, Type wt,
    int i, int n, Type bestw, QNode < Type > *E,
    QNode < Type > *&bestE, int bestx[], bool ch)
{// 将活结点加入到活结点队列 Q 中
  if (i == n) {// 可行叶结点
    if (wt == bestw) {
      // 当前最优载重量
      bestE = E;
      bestx[n] = ch;}
    return;}
  // 非叶结点
```

```
    QNode < Type >  * b;
    b = new QNode < Type > ;
    b -> weight = wt;
    b -> parent = E;
    b -> LChild = ch;
    Q.Add(b);
}
```

这样,算法就可以在搜索子集树的过程中保存当前已构造出的子集树中的路径指针,从而可在结束搜索后,从子集树中与最优值相应的结点处向根结点回溯,构造出相应的最优解。根据上述思想设计的新的队列式分支限界法可表述如下。算法结束后,bestx 中存放算法找到的最优解。

```
template < class Type >
Type MaxLoading(Type w[], Type c, int n, int bestx[])
{// 队列式分支限界法,返回最优载重量,bestx 返回最优解
    // 初始化
    Queue < QNode < Type > * > Q;   // 活结点队列
    Q.Add(0);                       // 同层结点尾部标志
    int i = 1;                      // 当前扩展结点所处的层
    Type Ew = 0,                    // 扩展结点所相应的载重量
      bestw = 0,                    // 当前最优载重量
      r = 0;                        // 剩余集装箱重量
    for (int j = 2; j < = n; j ++)
      r += w[i];
    QNode < Type >  * E = 0,        // 当前扩展结点
           * bestE;                 // 当前最优扩展结点
    // 搜索子集空间树
    while (true) {
      // 检查左儿子结点
      Type wt = Ew + w[i];
      if (wt < = c) {// 可行结点
        if (wt > bestw) bestw = wt;
        EnQueue(Q, wt, i, n, bestw, E, bestE, bestx, true);}
      // 检查右儿子结点
      if (Ew + r > bestw) EnQueue(Q, Ew, i, n,
                          bestw, E, bestE, bestx, false);
      Q.Delete(E);           // 取下一扩展结点
      if (!E) {              // 同层结点尾部
          if (Q.IsEmpty()) break;
          Q.Add(0);          // 同层结点尾部标志
          Q.Delete(E);       // 取下一扩展结点
```

```
        i++;            // 进入下一层
        r -= w[i];}     // 剩余集装箱重量
    Ew = E -> weight;  // 新扩展结点所相应的载重量
    }
  // 构造当前最优解
  for (int j = n - 1; j > 0; j--) {
      bestx[j] = bestE -> LChild;
      bestE = bestE -> parent;
  }
  return bestw;
}
```

4. 优先队列式分支限界法

解装载问题的优先队列式分支限界法将活结点表存储于一个最大优先队列中。活结点 x 在优先队列中的优先级定义为从根结点到结点 x 的路径所相应的载重量再加上剩余集装箱的重量之和。优先队列中优先级最大的活结点成为下一个扩展结点。优先队列中活结点 x 的优先级为 x.uweight。以结点 x 为根的子树中所有结点相应的路径的载重量不超过 x.uweight。子集树中叶结点所相应的载重量与其优先级相同。因此在优先队列式分支限界法中，一旦有一个叶结点成为当前扩展结点，则可以断言该叶结点所相应的解即为最优解，此时可终止算法。

上述策略可以用两种不同方式来实现。第一种方式在结点优先队列的每一个活结点中保存从解空间树的根结点到该活结点的路径，在算法确定了达到最优值的叶结点时，就在该叶结点处同时得到相应的最优解。第二种策略在算法的搜索进程中保存当前已构造出的部分解空间树，这样在算法确定了达到最优值的叶结点时，就可以在解空间树中从该叶结点开始向根结点回溯，构造出相应的最优解。在下面的算法中，我们采用第二种策略。

我们用一个元素类型为 HeapNode 的最大堆来表示活结点优先队列。其中 uweight 是活结点优先级（上界）；level 是活结点在子集树中所处的层序号；ptr 是指向活结点在子集树中相应结点的指针。子集空间树中结点类型为 bbnode。

```
template <class Type> class HeapNode;
class bbnode {
    friend void AddLiveNode(MaxHeap<HeapNode<int> > &, bbnode *,
                            int, bool, int);
    friend int MaxLoading(int *, int, int, int *);
    friend class AdjacencyGraph;
    private:
      bbnode * parent;      // 指向父结点的指针
      bool LChild;          // 左儿子结点标志
};

template <class Type>
class HeapNode {
    friend void AddLiveNode(MaxHeap<HeapNode<Type> > &, bbnode *,
```

```
                    Type, bool, int);
      friend Type MaxLoading(Type * , Type, int, int * );
      public:
        operator Type () const {return uweight;}
      private:
        bbnode * ptr;         // 指向活结点在子集树中相应结点的指针
        Type uweight;         // 活结点优先级(上界)
        int level;            // 活结点在子集树中所处的层序号
  };
```

在解装载问题的优先队列式分支限界法中,函数 AddLiveNode 以结点元素类型 bbnode 将一个新产生的活结点加入到子集树中,并以结点元素类型 HeapNode 将这个新结点插入到表示活结点优先队列的最大堆中。

```
template < class Type >
void AddLiveNode( MaxHeap < HeapNode < Type > > &H, bbnode * E,
                  Type wt, bool ch, int lev)
{// 将活结点加入到表示活结点优先队列的最大堆 H 中
    bbnode * b = new bbnode;
    b -> parent = E;
    b -> LChild = ch;
    HeapNode < Type > N;
    N.uweight = wt;
    N.level = lev;
    N.ptr = b;
    H.Insert(N);
}
```

函数 MaxLoading 具体实施对解空间的优先队列式分支限界搜索。在函数 MaxLoading 中,定义最大堆的容量为 1 000,即在算法运行期间,活结点优先队列最多可容纳 1 000 个活结点。第 $i+1$ 层结点的剩余重量 r[i] 定义为 $r[i]=\sum_{j=i+1}^{n} w[j]$。变量 E 指向子集树中当前扩展结点,Ew 是相应的重量。算法开始时,$i=1$,Ew $=0$,子集树的根结点是扩展结点。

while 循环体产生当前扩展结点的左右儿子结点。如果当前扩展结点的左儿子结点是可行结点,即它所相应的重量未超过船载容量,则将它加入到子集树的第 $i+1$ 层上,并插入最大堆。扩展结点的右儿子结点总是可行的,故直接插入子集树的最大堆中。接着算法从最大堆中取出最大元素作为下一个扩展结点。如果此时不存在下一个扩展结点,则相应的问题无可行解。如果下一个扩展结点是一个叶结点,即子集树中第 $n+1$ 层结点,则它相应的可行解为最优解。该最优解所相应的路径可由子集树中从该叶结点开始沿结点父指针逐步构造出来。具体算法可描述如下:

```
template < class Type >
Type MaxLoading( Type w[], Type c, int n, int bestx[])
{// 优先队列式分支限界法,返回最优载重量,bestx 返回最优解
    // 定义最大堆的容量为 1000
```

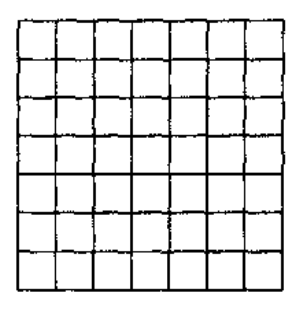
```
    MaxHeap < HeapNode < Type > > H(1000);
    // 定义剩余重量数组 r
    Type * r = new Type[n + 1];
    r[n] = 0;
    for (int j = n - 1; j > 0; j--)
        r[j] = r[j + 1] + w[j + 1];
    // 初始化
    int i = 1;              // 当前扩展结点所处的层
    bbnode * E = 0;         // 当前扩展结点
    Type Ew = 0;            // 扩展结点所相应的载重量
    // 搜索子集空间树
    while (i != n + 1) {// 非叶结点
        // 检查当前扩展结点的儿子结点
        if (Ew + w[i] <= c) {// 左儿子结点为可行结点
          AddLiveNode(H, E, Ew + w[i] + r[i], true, i + 1);}
        // 右儿子结点
        AddLiveNode(H, E, Ew + r[i], false, i + 1);
        // 取下一扩展结点
        HeapNode < Type > N;
        H.DeleteMax(N); // 非空
        i = N.level;
        E = N.ptr;
        Ew = N.uweight - r[i - 1];
        }
    // 构造当前最优解
    for (int j = n; j > 0; j--) {
        bestx[j] = E -> LChild;
        E = E -> parent;
        }
    return Ew;
}
```

算法中预先估计最大堆的容量(1 000)是由于用数组来实现最大堆所需要的。如果改用基于指针的优先队列实现方式则不必预先设置优先队列的容量。

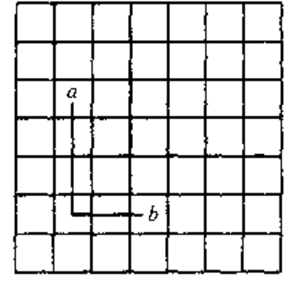
如果我们用变量 bestw 来记录当前子集树中可行结点所相应的重量的最大值,则当前活结点优先队列中可能包含某些结点的 uweight 值小于 bestw。易知以这些结点为根的子树中肯定不含最优解。如果不及时将这些结点从优先队列中删去,则一方面耗费优先队列的空间资源,另一方面增加执行优先队列的插入和删除操作的时间。为了避免产生这些无效活结点,可以在活结点插入优先队列前测试 uweight > bestw。通过测试的活结点才插入优先队列中。这样做可以避免产生一部分无效活结点。然而随着 bestw 不断增加,插入时还是有效的活结点,可能变成无效活结点。因此,为了及时删除由于 bestw 的增加而产生的无效活结点,即使 uweight < bestw 的活结点,要求优先队列除了支持 Insert, DeleteMax 运算外,还支持 DeleteMin 运算。这样的优先队列称为双端优先队列。有多种数据结构可有效地实现双端优先队列。

6.4 布线问题

印刷电路板将布线区域划分成 $n \times m$ 个方格阵列如图 6-3(a) 所示。精确的电路布线问题要求确定连接方格 a 的中点到方格 b 的中点的最短布线方案。在布线时，电路只能沿直线或直角布线，如图 6-3(b) 所示。为了避免线路相交，已布了线的方格做了封锁标记，其他线路不允许穿过被封锁的方格。



(a) 布线区域



(b) 沿直线或直角布线

图 6-3　印刷电路板布线方格阵列

下面我们讨论用队列式分支限界法来解布线问题。布线问题的解空间是一个图。解此问题的队列式分支限界法从起始位置 a 开始将它作为第一个扩展结点。与该扩展结点相邻并且可达的方格成为可行结点被加入到活结点队列中，并且将这些方格标记为 1，即从起始方格 a 到这些方格的距离为 1。接着，从活结点队列中取出队首结点作为下一个扩展结点，并将与当前扩展结点相邻且未标记过的方格标记为 2，并存入活结点队列。这个过程一直继续到算法搜索到目标方格 b 或活结点队列为空时为止。

在实现上述算法时，首先定义一个表示电路板上方格位置的类 Position，它的两个私有成员 row 和 col 分别表示方格所在的行和列。在电路板的任何一个方格处，布线可沿右、下、左、上 4 个方向进行。沿这 4 个方向的移动分别记为移动 0,1,2,3。在表6-1 中，offset[i].row 和 offset[i].col(i = 0,1,2,3) 分别给出沿这 4 个方向前进 1 步相对于当前方格的相对位移。

表6-1　移动方向的相对位移

移动 i	方向	offset[i].row	offset[i].col
0	右	0	1
1	下	1	0
2	左	0	-1
3	上	-1	0

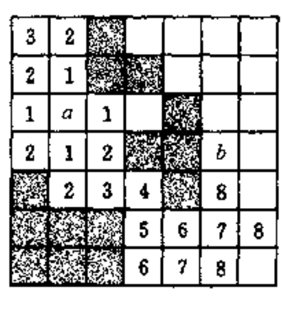
在实现上述算法时，我们用一个二维数组 grid 表示所给的方格阵列。初始时，grid[i][j] = 0，表示该方格允许布线，而 grid[i][j] = 1 表示该方格被封锁，不允许布线。为了便于处理方格边界的情况，算法在所给方格阵列四周设置一道“围墙”，即增设标记为“1” 的附加方格。算法开始时测试初始方格与目标方格是否相同。如果这 2 个方格相同则不必计算，直接返回最短距离 0，否则算法设置方格阵列的“围墙”，初始化位移矩阵 offset。算法将起始位置的距离标记为 2。由于数字 0 和 1 用于表示方格的开放或封锁状态，所以在表示距离时不用这 2 个数字，因而将距离的值都加 2。实际距离应为标记距离减 2。算法从起始位置 start 开始，标记所有标记

距离为 3 的方格并存入活结点队列,然后依次标记所有标记距离为 4,5,… 的方格,直至到达目标方格 finish 或活结点队列为空时为止。具体算法可描述如下:

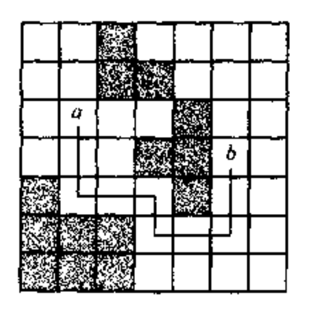
```
bool FindPath (Position start, Position finish,
            int& PathLen, Position * &path)
{// 计算从起始位置 start 到目标位置 finish 的最短布线路径
// 找到最短布线路径则返回 true,否则返回 false
    if ((start.row == finish.row) &&
       (start.col == finish.col))
        {PathLen = 0; return true;}
    // 设置方格阵列"围墙"
    for (int i = 0; i <= m + 1; i++)
       grid[0][i] = grid[n + 1][i] = 1; // 顶部和底部
    for (int i = 0; i <= n + 1; i++)
       grid[i][0] = grid[i][m + 1] = 1; // 左翼和右翼
    // 初始化相对位移
    Position offset[4];
    offset[0].row = 0; offset[0].col = 1; // 右
    offset[1].row = 1; offset[1].col = 0; // 下
    offset[2].row = 0; offset[2].col = -1; // 左
    offset[3].row = -1; offset[3].col = 0; // 上
    int NumOfNbrs = 4; // 相邻方格数
    Position here, nbr;
    here.row = start.row;
    here.col = start.col;
    grid[start.row][start.col] = 2;
    // 标记可达方格位置
    LinkedQueue < Position > Q;
    do {// 标记可达相邻方格
      for (int i = 0; i < NumOfNbrs; i++) {
          nbr.row = here.row + offset[i].row;
          nbr.col = here.col + offset[i].col;
          if (grid[nbr.row][nbr.col] == 0) {
             // 该方格未标记
             grid[nbr.row][nbr.col]
                = grid[here.row][here.col] + 1;
             if ((nbr.row == finish.row) &&
               (nbr.col == finish.col)) break; // 完成布线
           Q.Add(nbr);}
           }
     // 是否到达目标位置 finish?
     if ((nbr.row == finish.row) &&
       (nbr.col == finish.col)) break; // 完成布线
     // 活结点队列是否非空
```

```
    if (Q.IsEmpty()) return false; // 无解
    Q.Delete(here); // 取下一个扩展结点
  } while(true);
  // 构造最短布线路径
  PathLen = grid[finish.row][finish.col] - 2;
  path = new Position [PathLen];
  // 从目标位置 finish 开始向起始位置回溯
  here = finish;
  for (int j = PathLen - 1; j >= 0; j--) {
    path[j] = here;
    // 找前驱位置
    for (int i = 0; i < NumOfNbrs; i++) {
      nbr.row = here.row + offset[i].row;
      nbr.col = here.col + offset[i].col;
      if (grid[nbr.row][nbr.col] == j + 2) break;
      }
    here = nbr; // 向前移动
    }
  return true;
}
```

图 6-4 是在一个 7×7 方格阵列中布线的例子。其中起始位置是 $a = (3,2)$，目标位置是 $b = (4,6)$，阴影方格表示被封锁的方格。当算法搜索到目标方格 b 时，将目标方格 b 标记为从起始位置 a 到 b 的最短距离。本例中，a 到 b 的最短距离是 9。要构造出与最短距离相应的最短路径，我们从目标方格开始向起始方格方向回溯，逐步构造出最优解。每次向标记距离比当前方格标记距离少 1 的相邻方格移动，直至到达起始方格时为止。在图6-4(a) 中，我们从目标方格 b 移到(5,6)，然后移至(6,6)，…，最终移至起始方格 a，得到相应的最短路径如图 6-4(b) 所示。



(a) 标记距离



(b) 最短布线路径

图 6-4　布线算法示例

由于每个方格成为活结点进入活结点队列最多 1 次，因此活结点队列中最多只处理 $O(mn)$ 个活结点。扩展每个结点需 $O(1)$ 时间，因此算法共耗时 $O(mn)$。构造相应的最短距离需要 $O(L)$ 时间，其中 L 是最短布线路径的长度。

6.5 0-1 背包问题

在解0-1背包问题的优先队列式分支限界法中，活结点优先队列中结点元素 N 的优先级由该结点的上界函数Bound计算出的值uprofit给出。该上界函数已在讨论解0-1背包问题的回溯法时讨论过。子集树中以结点 N 为根的子树中任一结点的价值不超过 N.profit。因此我们用一个最大堆来实现活结点优先队列，堆中元素类型为 HeapNode，其私有成员有 uprofit，profit，weight，level和 ptr。对于任意一个活结点 N，N.weight是结点 N所相应的重量；N.profit是 N所相应的价值；N.uprofit 是结点 N 的价值上界，最大堆以这个值作为优先级 子集空间树中结点类型为 bbnode。

```
class Object {
   friend int Knapsack(int * , int * , int, int, int * );
   public:
     int operator < = (Object a) const
     {return (d > = a.d);}
   private:
     int ID;
     float d; // 单位重量价值
};

template < class Typew, class Typep > class Knap;
class bbnode {
   friend Knap < int,int > ;
   friend int Knapsack(int * , int * , int, int, int * );
   private:
     bbnode * parent;   // 指向父结点的指针
     bool LChild;       // 左儿子结点标志
};

template < class Typew, class Typep >
class HeapNode {
   friend Knap < Typew,Typep > ;
   public:
     operator Typep () const {return uprofit;}
   private:
     Typep uprofit,       // 结点的价值上界
           profit;        // 结点所相应的价值
     Typew weight;        // 结点所相应的重量
     int level;           // 活结点在子集树中所处的层序号
     bbnode * ptr;        // 指向活结点在子集树中相应结点的指针
};
```

这里用到的类 Knap 与解0-1背包问题的回溯法中用到的类 Knap 十分相似。它们的区别是新的类中没有成员变量 bestp,而增加了新的成员 bestx。bestx[i] = 1 当且仅当最优解含有物品 i。

```
template < class Typew, class Typep >
class Knap {
    friend Typep Knapsack(Typep *, Typew *, Typew, int, int *);
    public:
      Typep MaxKnapsack();
    private:
      MaxHeap < HeapNode < Typep, Typew > > * H;
      Typep Bound(int i);
      void AddLiveNode(Typep up, Typep cp, Typew cw, bool ch, int level);
      bbnode * E;         // 指向扩展结点的指针
      Typew c;            // 背包容量
      int n;              // 物品总数
      Typew * w;          // 物品重量数组
      Typep * p;          // 物品价值数组
      Typew cw;           // 当前装包重量
      Typep cp;           // 当前装包价值
      int * bestx;        // 最优解
};
```

上界函数 Bound 计算结点所相应价值的上界。

```
template < class Typew, class Typep >
Typep Knap < Typew, Typep > ::Bound(int i)
{// 计算结点所相应价值的上界
    Typew cleft = c - cw;      // 剩余容量
    Typep b = cp;              // 价值上界
    // 以物品单位重量价值递减序装填剩余容量
    while (i <= n && w[i] <= cleft) {
      cleft -= w[i];
      b += p[i];
      i++;
      }
    // 装填剩余容量装满背包
    if (i <= n) b += p[i]/w[i] * cleft;
    return b;
}
```

函数 AddLiveNode 将一个新的活结点插入到子集树和优先队列中。

```
template < class Typep, class Typew >
void Knap < Typep, Typew > ::AddLiveNode(Typep up,
```

```
                Typep cp, Typew cw, bool ch, int lev)
{// 将一个新的活结点插入到子集树和最大堆 H 中
    bbnode *b = new bbnode;
    b->parent = E;
    b->LChild = ch;
    HeapNode<Typep, Typew> N;
    N.uprofit = up;
    N.profit = cp;
    N.weight = cw;
    N.level = lev;
    N.ptr = b;
    H->Insert(N);
}
```

函数 MaxKnapsack 实施对子集树的优先队列式分支限界搜索。其中假定各物品依其单位重量价值从大到小排好序。相应的排序过程可在算法的预处理部分完成。

算法中 E 是当前扩展结点;cw 是该结点所相应的重量;cp 是相应的价值;up 是价值上界。算法的 while 循环不断扩展结点,直到子集树的一个叶结点成为扩展结点时为止。此时优先队列中所有活结点的价值上界均不超过该叶结点的价值。因此该叶结点相应的解为问题的最优解。

在 while 循环内部,算法首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点是可行结点,则将它加入到子集树和活结点优先队列中。当前扩展结点的右儿子结点一定是可行结点,仅当右儿子结点满足上界约束时才将它加入子集树和活结点优先队列。算法 MaxKnapsack 具体描述如下:

```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::MaxKnapsack()
{// 优先队列式分支限界法,返回最大价值,bestx 返回最优解
    // 定义最大堆的容量为 1000
    H = new MaxHeap<HeapNode<Typep, Typew> > (1000);
    // 为 bestx 分配存储空间
    bestx = new int [n+1];
    // 初始化
    int i = 1;
    E = 0;
    cw = cp = 0;
    Typep bestp = 0;          // 当前最优值
    Typep up = Bound(1);      // 价值上界
    // 搜索子集空间树
    while (i != n+1) {// 非叶结点
      // 检查当前扩展结点的左儿子结点
      Typew wt = cw + w[i];
      if (wt <= c) {// 左儿子结点为可行结点
```

```
        if (cp + p[i] > bestp) bestp = cp + p[i];
        AddLiveNode(up, cp + p[i], cw + w[i], true, i + 1);}
        up = Bound(i + 1);
      // 检查当前扩展结点的右儿子结点
      if (up >= bestp) // 右子树可能含最优解
          AddLiveNode(up, cp, cw, false, i + 1);
      // 取下一扩展结点
      HeapNode< Typep, Typew > N;
      H->DeleteMax(N);
      E = N.ptr;
      cw = N.weight;
      cp = N.profit;
      up = N.uprofit;
      i = N.level;
      }
    // 构造当前最优解
    for (int j = n; j > 0; j--) {
      bestx[j] = E->LChild;
      E = E->parent;
      }
    return cp;
}
```

下面的函数 Knapsack 完成对输入数据的预处理。其主要任务是将各物品依其单位重量价值从大到小排好序。然后调用函数 MaxKnapsack 完成对子集树的优先队列式分支限界搜索。

```
template< class Typew, class Typep >
Typep Knapsack(Typep p[], Typew w[], Typew c, int n, int bestx[])
{// 返回最大价值,bestx 返回最优解
    // 初始化
    Typew W = 0;  // 装包物品重量
    Typep P = 0;  // 装包物品价值
    // 定义依单位重量价值排序的物品数组
    Object * Q = new Object [n];
    for (int i = 1; i <= n; i++) {
      // 单位重量价值数组
      Q[i-1].ID = i;
      Q[i-1].d = 1.0 * p[i]/w[i];
      P += p[i];
      W += w[i];
      }
    if (W <= c) return P; // 所有物品装包
    // 依单位重量价值排序
```

```
    Sort(Q,n);
    // 创建类 Knap 的数据成员
    Knap< Typew, Typep> K;
    K.p = new Typep [n+1];
    K.w = new Typew [n+1];
    for (int i = 1; i <= n; i++) {
      K.p[i] = p[Q[i-1].ID];
      K.w[i] = w[Q[i-1].ID];
      }
    K.cp = 0;
    K.cw = 0;
    K.c = c;
    K.n = n;
    // 调用 MaxKnapsack 求问题的最优解
    Typep bestp = K.MaxKnapsack();
    for (int j = 1 ; j <= n; j++)
      bestx[Q[j-1].ID] = K.bestx[j];
    delete [] Q;
    delete [] K.w;
    delete [] K.p;
    delete [] K.bestx;
    return bestp;
}
```

6.6 最大团问题

最大团问题的解空间树也是一棵子集树。解最大团问题的优先队列式分支限界法与解装载问题的优先队列式分支限界法相似。算法构造的解空间树中结点类型是 bbnode;活结点优先队列中元素类型为 CliqueNode。每一个 CliqueNode 类型的结点都以变量 cn 表示与该结点相应的团的顶点数;un 表示该结点为根的子树中最大顶点数的上界;level 表示结点在子集空间树中所处的层次;ch 是左、右儿子结点标记。当 ch = 1 时,表示该结点是其父结点的左儿子结点,当 ch = 0 时是右儿子结点。ptr 是指向解空间树中相应结点的指针。我们用 cn + n - level + 1 作为顶点数上界 un 的值。由此,可省去一个变量 cn 或 level,因为从 un 的值可推出省去变量的值。un 实际上也是优先队列中元素的优先级。算法总是从活结点优先队列中抽取具有最大 un 值的元素作为下一个扩展元素。

```
class bbnode {
  friend class Clique;
  private:
    bbnode * parent; // 指向父结点的指针
    bool LChild;     // 左儿子结点标志
};
```

```
class CliqueNode {
   friend class Clique;
   public:
     operator int () const {return un;}
   private:
     int cn,            // 当前团的顶点数
         un,            // 当前团最大顶点数的上界
         level;            // 结点在子集空间树中所处的层次
     bbnode * ptr;       // 指向活结点在子集树中相应结点的指针
};
```

在具体实现时，用邻接矩阵表示所给的图 G。在类 Clique 中用一个二维数组 a 存储图 G 的邻接矩阵。

```
class Clique {
    friend void main(void);
    public: int BBMaxClique(int []);
    private:
      void AddLiveNode(MaxHeap < CliqueNode >  &H,
            int cn, int un, int level, bbnode E[], bool ch);
      int * * a,   // 图 G 的邻接矩阵
            n;     // 图 G 的顶点数
};
```

算法中函数 AddLiveNode 的功能是将当前构造出的活结点加入到子集空间树中并插入活结点优先队列中。

```
void Clique::AddLiveNode(MaxHeap < CliqueNode >
      &H, int cn, int un, int level, bbnode E[], bool ch)
  {// 将活结点加入到子集空间树中并插入最大堆中
     bbnode * b = new bbnode;
     b -> parent = E;
     b -> LChild = ch;
     CliqueNode N;
     N.cn = cn;
     N.ptr = b;
     N.un = un;
     N.level = level;
     H.Insert(N);
  }
```

函数 BBMaxClique 具体实现对子集解空间树的最大优先队列式分支限界搜索。子集树的根结点是初始扩展结点。对于这个特殊的扩展结点，其 cn 的值为 0。变量 i 用于表示当前扩展结点在解空间树中所处的层次。因此，初始时扩展结点所相应的 i 值为 1，当前最大团的顶点数

存储于变量 bestn 中。

在 while 循环中，我们不断从活结点优先队列中抽取当前扩展结点并实施对该结点的扩展。while循环的终止条件是遇到子集树中的一个叶结点（即 $n+1$ 层结点）成为当前扩展结点。对于子集树中的一个叶结点，我们有 un = cn。此时活结点优先队列中剩余结点的 un 值均不超过当前扩展结点的 un 值，从而进一步搜索不可能得到更大的团，此时算法已找到一个最优解。

算法在扩展一个内部结点时，首先考察其左儿子结点。在左儿子结点处，将顶点 i 加入到当前团中，并检查该顶点与当前团中其他顶点之间是否有边相连。当顶点 i 与当前团中所有顶点之间都有边相连，则相应的左儿子结点是可行结点，否则就不是可行结点。为了检测左儿子结点的可行性，算法从当前扩展结点开始向根结点回溯，确定当前团中的顶点，同时检查当前团中的顶点与顶点 i 的连接情况。如果左儿子结点是一个可行结点，则将它加入到子集树中并插入活结点优先队列。接着算法继续考察当前扩展结点的右儿子结点。当 un > bestn 时，右子树中可能含有最优解，此时将右儿子结点加入到子集树中并插入到活结点优先队列中。

由于每一个图都有最大团，因此在从最大堆中抽取极大元素时不必测试堆是否为空。算法的 while 循环仅当遇到一个叶结点时退出。

```
int Clique::BBMaxClique(int bestx[])
{// 解最大团问题的优先队列式分支限界法
  // 定义最大堆的容量为 1000
  MaxHeap < CliqueNode > H(1000);
  // 初始化
  bbnode * E = 0;
  int i = 1,
      cn = 0,
      bestn = 0;
  // 搜索子集空间树
  while (i ! = n + 1) {// 非叶结点
      // 检查顶点 i 与当前团中其他顶点之间是否有边相连
      bool OK = true;
      bbnode * B = E;
      for (int j = i - 1; j > 0; B = B -> parent, j --)
        if (B -> LChild && a[i][j] = = 0) {
            OK = false;
            break;}
      if (OK) {// 左儿子结点为可行结点
        if (cn + 1 > bestn) bestn = cn + 1;
        AddLiveNode(H, cn + 1, cn + n - i + 1, i + 1, E, true);}
      if (cn + n - i > = bestn)
        // 右子树可能含最优解
        AddLiveNode(H, cn, cn + n - i, i + 1, E, false);
      // 取下一扩展结点
      CliqueNode N;
      H.DeleteMax(N); // 堆非空
      E = N.ptr;
```

```
        cn = N.cn;
        i = N.level;
        }
    // 构造当前最优解
    for (int j = n; j > 0; j--) {
        bestx[j] = E -> LChild;
        E = E -> parent;
        }
    return bestn;
}
```

6.7 旅行售货员问题

旅行售货员问题的解空间树是一棵排列树。与前面关于子集树的讨论类似,实现对排列树搜索的优先队列式分支限界法也可以有两种不同的实现方式。其一是仅使用一个优先队列来存储活结点。优先队列中的每个活结点都存储从根到该活结点的相应路径。另一种实现方式是用优先队列来存储活结点,并同时存储当前已构造出的部分排列树。在这种实现方式下,优先队列中的活结点就不必再存储从根到该活结点的相应路径。这条路径可在必要时从存储的部分排列树中获得。在下面的讨论中我们采用第一种实现方式。

我们用邻接矩阵表示所给的图 G。在类 Traveling 中用一个二维数组 a 存储图 G 的邻接矩阵。

```
template < class Type >
class Traveling {
   friend void main(void);
   public:
     Type BBTSP(int v[]);
   private:
     int n;          // 图 G 的顶点数
     Type * * a,     // 图 G 的邻接矩阵
          NoEdge,    // 图 G 的无边标志
          cc,        // 当前费用
          bestc;     // 当前最小费用
};
```

由于我们要找的是最小费用旅行售货员回路,所以我们选用最小堆表示活结点优先队列。最小堆中元素的类型为 MinHeapNode。该类型结点包含域 x,用于记录当前解;s 表示结点在排列树中的层次,从排列树的根结点到该结点的路径为 x[0:s],需要进一步搜索的顶点是 x[s + 1:n - 1]。cc 表示当前费用,lcost 是子树费用的下界,rcost 是 x[s:n - 1] 中顶点最小出边费用和。具体算法可描述如下:

```
template < class Type >
```

```
class MinHeapNode {
   friend Traveling < Type >;
   public:
     operator Type () const {return lcost;}
   private:
     Type lcost,      // 子树费用的下界
          cc,         // 当前费用
          rcost;      // x[s:n - 1]中顶点最小出边费用和
     int   s,         // 根结点到当前结点的路径为 x[0:s]
          * x;        // 需要进一步搜索的顶点是 x[s + 1:n - 1]
};
```

算法开始时创建一个容量为 1 000 的最小堆,用于表示活结点优先队列。堆中每个结点的 lcost 值是优先队列的优先级。接着计算出图中每个顶点的最小费用出边并用 Minout 记录。如果所给的有向图中某个顶点没有出边,则该图不可能有回路,算法即告结束。如果每个顶点都有出边,则根据计算出的 Minout 作算法初始化。算法的第 1 个扩展结点是排列树中根结点的惟一儿子结点(图 5-3 中结点 B)。在该结点处,已确定的回路中惟一顶点为顶点 1。因此,初始时有 $s = 0, x[0] = 1, x[1:n-1] = (2,3,\cdots,n)$, cc = 0 且 rcost = $\sum_{i=s}^{n}$ Minout[i]。算法中用 bestc 记录当前最优值,初始时还没有找到回路,故 bestc = NoEdge。

```
template < class Type >
Type Traveling < Type >::BBTSP(int v[])
{// 解旅行售货员问题的优先队列式分支限界法
   // 定义最小堆的容量为 1000
   MinHeap < MinHeapNode < Type > > H(1000);
   Type * MinOut = new Type [n + 1];
   // 计算 MinOut[i] = 顶点 i 的最小出边费用
   Type MinSum = 0;  // 最小出边费用和
   for (int i = 1; i <= n; i ++) {
     Type Min = NoEdge;
     for (int j = 1; j <= n; j ++)
       if (a[i][j] != NoEdge &&
                (a[i][j] < Min || Min == NoEdge))
            Min = a[i][j];
     if (Min == NoEdge) return NoEdge; // 无回路
     MinOut[i] = Min;
     MinSum += Min;
     }
   // 初始化
   MinHeapNode < Type > E;
   E.x = new int [n];
   for (int i = 0; i < n; i ++)
      E.x[i] = i + 1;
```

```
E.s = 0;
E.cc = 0;
E.rcost = MinSum;
Type bestc = NoEdge;
// 搜索排列空间树
while (E.s < n - 1) {// 非叶结点
  if (E.s == n - 2) {// 当前扩展结点是叶结点的父结点
     // 再加 2 条边构成回路
     // 所构成回路是否优于当前最优解
     if (a[E.x[n - 2]][E.x[n - 1]] != NoEdge &&
          a[E.x[n - 1]][1] != NoEdge && (E.cc +
          a[E.x[n - 2]][E.x[n - 1]] + a[E.x[n - 1]][1]
          < bestc || bestc == NoEdge)) {
          // 费用更小的回路
          bestc = E.cc + a[E.x[n - 2]][E.x[n - 1]] + a[E.x[n - 1]][1];
          E.cc = bestc;
          E.lcost = bestc;
          E.s ++;
          H.Insert(E);}
     else delete [] E.x;} // 舍弃扩展结点
  else {// 产生当前扩展结点的儿子结点
     for (int i = E.s + 1; i < n; i ++)
       if (a[E.x[E.s]][E.x[i]] != NoEdge) {
           // 可行儿子结点
           Type cc = E.cc + a[E.x[E.s]][E.x[i]];
           Type rcost = E.rcost - MinOut[E.x[E.s]];
           Type b = cc + rcost; // 下界
           if (b < bestc || bestc == NoEdge) {
                 // 子树可能含最优解
                 // 结点插入最小堆
                 MinHeapNode < Type > N;
                 N.x = new int [n];
                 for (int j = 0; j < n; j ++)
                    N.x[j] = E.x[j];
                 N.x[E.s + 1] = E.x[i];
                 N.x[i] = E.x[E.s + 1];
                 N.cc = cc;
                 N.s = E.s + 1;
                 N.lcost = b;
                 N.rcost = rcost;
                 H.Insert(N);}
     }
  delete [] E.x;} // 完成结点扩展
try {H.DeleteMin(E);}            // 取下一扩展结点
```

```
    catch (OutOfBounds) {break;}    // 堆已空
    }
  if (bestc == NoEdge) return NoEdge; // 无回路
  // 将最优解复制到 v[1:n]
  for (int i = 0; i < n; i++)
     v[i + 1] = E.x[i];
  while (true) {// 释放最小堆中所有结点
    delete [] E.x;
    try {H.DeleteMin(E);}
    catch (OutOfBounds) {break;}
    }
  return bestc;
}
```

算法中 while 循环的终止条件是排列树的一个叶结点成为当前扩展结点。当 $s = n - 1$ 时，已找到的回路前缀是 $x[0:n-1]$，它已包含图 G 的所有 n 个顶点。因此，当 $s = n - 1$ 时，相应的扩展结点表示一个叶结点。此时该叶结点所相应的回路的费用等于 cc 和 lcost 的值。剩余的活结点的 lcost 值不小于已找到的回路的费用。它们都不可能导致费用更小的回路。因此已找到的叶结点所相应的回路是一个最小费用旅行售货员回路，算法可以结束。

算法的 while 循环体完成对排列树内部结点的扩展。对于当前扩展结点，算法分两种情况进行处理。首先考虑 $s = n - 2$ 的情形。此时当前扩展结点是排列树中某个叶结点的父结点。如果该叶结点相应一条可行回路且费用小于当前最小费用，则将该叶结点插入到优先队列中，否则舍去该叶结点。

当 $s < n - 2$ 时，算法依次产生当前扩展结点的所有儿子结点。由于当前扩展结点所相应的路径是 $x[0:s]$，其可行儿子结点是从剩余顶点 $x[s+1:n-1]$ 中选取的顶点 $x[i]$，且 $(x[s], x[i])$ 是所给有向图 G 中的一条边。对于当前扩展结点的每一个可行儿子结点，计算出其前缀 $(x[0:s], x[i])$ 的费用 cc 和相应的下界 lcost。当 lcost < bestc 时，将这个可行儿子结点插入到活结点优先队列中。

在所给的有向图没有回路时，算法返回 NoEdge。否则返回找到的最小费用，相应的最优解由数组 v 给出。

6.8 电路板排列问题

电路板排列问题的解空间树也是一棵排列树。我们采用优先队列式分支限界法找出所给电路板的最小密度布局。算法中用一个最小堆来表示活结点优先队列。最小堆中元素类型是 BoardNode。每一个 BoardNode 类型的结点包含域 x，用来表示结点所相应的电路板排列；s 表示该结点已确定的电路板排列 $x[1:s]$；cd 表示当前密度；$now[j]$ 表示 $x[1:s]$ 中所含连接块 j 中的电路板数。具体算法描述如下：

```
class BoardNode {
   friend int BBArrangement(int **, int, int, int* &);
   public:
     operator int () const {return cd;}
```

```
private:
    int * x,        // x[1:n] 记录电路板排列
          s,        // x[1:s] 是当前结点所相应的部分排列
          cd,       // x[1:s] 的密度
          * now;    // now[j] 是 x[1:s] 所含连接块 j 中电路板数
};
```

函数 BBArrangement 是解电路板排列问题的优先队列式分支限界法的主体。算法开始时，将排列树的根结点置为当前扩展结点。在初始扩展结点处还没有选定电路板，故 $s = 0$，$cd = 0$，$now[i] = 0$，$1 \leq i \leq n$。且数组 x 初始化为单位排列。数组 total 初始化为 total[i] 等于连接块 i 所含电路板数。bestd 表示当前最小密度，bestx 是相应的最优解。

算法的 do-while 循环完成对排列树内部结点的有序扩展。在 do-while 循环体内算法依次从活结点优先队列中取出具有最小 cd 值的结点作为当前扩展结点，并加以扩展。如果当前扩展结点的 cd ≥ bestd，则优先队列中其余活结点都不可能导致最优解，此时算法结束。

算法将当前扩展结点分为两种情形处理。首先考虑 $s = n - 1$ 的情形，此时已排定 $n - 1$ 块电路板，故当前扩展结点是排列树中的一个叶结点的父结点。x 表示相应于该叶结点的电路板排列。计算出与 x 相应的密度并在必要时更新当前最优值 bestd 和相应的当前最优解 bestx。

当 $s < n - 1$ 时，算法依次产生当前扩展结点的所有儿子结点。对于当前扩展结点的每一个儿子结点 N，计算出其相应的密度 N.cd。当 N.cd < bestd 时，将该儿子结点 N 插入到活结点优先队列中。而当 N.cd ≥ bestd 时，以 N 为根的子树中不可能有比当前最优解 bestx 更好的解，故可将结点 N 舍去。

```
int BBArrangement(int * * B, int n, int m, int * &bestx)
{// 解电路板排列问题的优先队列式分支限界法
    MinHeap < BoardNode > H(1000); // 活结点最小堆
    // 初始化
    BoardNode E;
    E.x = new int [n + 1];
    E.s = 0;
    E.cd = 0;
    E.now = new int [m + 1];
    int * total = new int [m + 1];
    // now[i] = x[1:s] 所含连接块 i 中电路板数
    // total[i] = 连接块 i 中电路板数
    for (int i = 1; i < = m; i ++) {
        total[i] = 0;
        E.now[i] = 0;
        }
    for (int i = 1; i < = n; i ++) {
      E.x[i] = i; // 初始排列为 12345...n
      for (int j = 1; j < = m; j ++)
          total[j] + = B[i][j]; // 连接块 j 中电路板数
      }
```

```
int bestd = m + 1; // 当前最小密度
bestx = 0;
do {// 结点扩展
  if (E.s == n - 1) {// 仅一个儿子结点
      int ld = 0; // 最后一块电路板的密度
      for (int j = 1; j <= m; j++)
          ld += B[E.x[n]][j];
      if (ld < bestd) {// 密度更小的电路板排列
         delete [] bestx;
         bestx = E.x;
         bestd = max(ld, E.cd);
         }
   else delete [] E.x;
   delete [] E.now;}
else {// 产生当前扩展结点的所有儿子结点
for (int i = E.s + 1; i <= n; i++) {
   BoardNode N;
   N.now = new int [m + 1];
   for (int j = 1; j <= m; j++)
      // 新插入的电路板
      N.now[j] = E.now[j] + B[E.x[i]][j];
  int ld = 0; // 新插入电路板的密度
  for (int j = 1; j <= m; j++)
      if (N.now[j] > 0 && total[j] != N.now[j])
          ld++;
  N.cd = max(ld, E.cd);
  if (N.cd < bestd) {// 可能产生更好的叶结点
      N.x = new int [n + 1];
      N.s = E.s + 1;
      for (int j = 1; j <= n; j++)
          N.x[j] = E.x[j];
      N.x[N.s] = E.x[i];
      N.x[i] = E.x[N.s];
      H.Insert(N);}
  else delete [] N.now;}
  delete [] E.x;} // 完成当前结点扩展
try {H.DeleteMin(E);} // 取下一扩展结点
catch (OutOfBounds) {return bestd;} // 无扩展结点
} while (E.cd < bestd);
// 释放最小堆中所有结点
do {delete [] E.x;
   delete [] E.now;
   try {H.DeleteMin(E);}
   catch (...) {break;}
```

```
    } while (true);
  return bestd;
```

6.9 批处理作业调度

由5.3中关于批处理作业调度的回溯法分析可知，批处理作业调度问题的解空间树是一棵排列树。在作业调度问题相应的排列空间树中，每一个结点 E 都对应于一个已安排的作业集 $M \subseteq \{1,2,\cdots,n\}$。以该结点为根的子树中所含叶结点的完成时间和可表示为：

$$f = \sum_{i \in M} F_{2i} + \sum_{i \notin M} F_{2i}$$

设 $|M| = r$，且 L 是以结点 E 为根的子树中的一个叶结点，相应的作业调度为 $\{p_k, k = 1,2,\cdots,n\}$，其中，$p_k$ 是第 k 个安排的作业。如果从结点 E 开始到叶结点 L 的路上，每一个作业 p_k 在机器1上完成处理后都能立即在机器2上开始处理，即从 p_{r+1} 开始，机器1没有空闲时间，则对于该叶结点 L 有

$$\sum_{i \notin M} F_{2i} = \sum_{k=r+1}^{n} [F_{1p_r} + (n-k+1)t_{1p_k} + t_{2p_k}] = S_1$$

如果不能做到上面这一点，则 S_1 只会增加，从而我们有 $\sum_{i \notin M} F_{2i} \geqslant S_1$。

类似地，如果从结点 E 开始到结点 L 的路上，从作业 p_{r+1} 开始，机器2没有空闲时间，则

$$\sum_{i \notin M} F_{2i} \geqslant \sum_{k=r+1}^{n} [\max(F_{2p_r}, F_{1p_r} + \min_{i \notin M} t_{1i}) + (n-k+1)t_{2p_k}] = S_2$$

同理可知，S_2 是 $\sum_{i \notin M} F_{2i}$ 的一个下界。由此，我们得到在结点 E 处相应子树中叶结点完成时间和的下界是

$$f \geqslant \sum_{i \in M} F_{2i} + \max\{S_1, S_2\}$$

其中，S_1 与 S_2 的计算依赖于叶结点 L 相应的作业调度 $\{p_k, k = 1,2,\cdots,n\}$。注意到如果选择 p_k，使 t_{1p_k} 在 $k \geqslant r+1$ 时依非减序排列，则 S_1 取得极小值 $\hat{S}_1$。同理如果选择 p_k 使 t_{2p_k} 依非减序排列，则 S_2 取得极小值 $\hat{S}_2$。因此 $S_1 \geqslant \hat{S}_1$，$S_2 \geqslant \hat{S}_2$，且 $\hat{S}_1$ 和 $\hat{S}_2$ 与叶结点的调度无关。从而我们有

$$f \geqslant \sum_{i \in M} F_{2i} + \max\{\hat{S}_1, \hat{S}_2\}$$

这可以作为优先队列式分支限界法中的限界函数。

算法中用一个最小堆来表示活结点优先队列。最小堆中元素类型是 MinHeapNode。每一个 MinHeapNode 类型的结点包含域 x，用来表示结点所相应的作业调度。s 表示该结点已安排的作业是 x[1:s]。f1 表示当前已安排的作业在机器1上的最后完成时间；f2 表示当前已安排的作业在机器2上的最后完成时间；sf2 表示当前已安排的作业在机器2上的完成时间和；bb表示当前完成时间和的下界。函数 Init 完成最小堆结点初始化；函数 NewNode 产生最小堆新结点。

```
class Flowshop;
class MinHeapNode {
```

```
        friend Flowshop;
        public:
          operator int () const {return bb;}
        private:
          void Init(int),
               NewNode(MinHeapNode,int,int,int,int);
          int  s,        // 已安排作业数
               f1,       // 机器 1 上最后完成时间
               f2,       // 机器 2 上最后完成时间
               sf2,      // 当前机器 2 上的完成时间和
               bb,       // 当前完成时间和下界
               * x;      // 当前作业调度
};

void MinHeapNode::Init(int n)
{ // 最小堆结点初始化
  x = new int [n];
  for (int i = 0; i < n; i ++)
    x[i] = i;
  s = 0;
  f1 = 0;
  f2 = 0;
  sf2 = 0;
  bb = 0;
}

void MinHeapNode::NewNode(MinHeapNode E,int Ef1,
                                  int Ef2,int Ebb,int n)
{ // 最小堆新结点
  x = new int [n];
  for (int i = 0; i < n; i ++)
      x[i] = E.x[i];
  f1 = Ef1;
  f2 = Ef2;
  sf2 = E.sf2 + f2;
  bb = Ebb;
  s = E.s + 1;
}
```

在具体实现时，用一个二维数组 M 表示所给的 n 个作业在机器 1 和机器 2 所需的处理时间。在类 Flowshop 中用一个二维数组 b 存储排好序的作业处理时间。数组 a 表示数组 M 和 b 的对应关系。bestc 记录当前最小完成时间和，bestx 记录相应的当前最优解。函数 Sort 实现对各作

业在机器 1 和 2 上所需时间排序。函数 Bound 用于计算完成时间和下界。

```
class Flowshop {
  friend void main(void);
  public:
    int BBFlow(void);
  private:
    int Bound(MinHeapNode,int &,int &,bool * *);
    void Sort(void);
    int   n,            // 作业数
          * * M,        // 各作业所需的处理时间数组
          * * b,        // 各作业所需的处理时间排序数组
          * * a,        // 数组 M 和 b 的对应关系数组
          * bestx,      // 最优解
          bestc;        // 最小完成时间和
    bool * * y;         // 工作数组
};
```

```
void Flowshop::Sort(void)
{// 对各作业在机器 1 和 2 上所需时间排序
    int *c = new int [n];
    for (int j = 0;j < 2;j++) {
      for (int i = 0;i < n;i++) {
        b[i][j] = M[i][j];
        c[i] = i;}
        for (int i = 0;i < n - 1;i++)
          for (int k = n - 1; k > i;k--)
            if (b[k][j] < b[k - 1][j]) {
              Swap(b[k][j], b[k - 1][j]);
              Swap(c[k], c[k - 1]);}
        for (int i = 0;i < n;i++) a[c[i]][j] = i; }
    delete [] c;
}
```

```
int Flowshop::Bound(MinHeapNode E,int & f1,int & f2,bool * * y)
{// 计算完成时间和下界
  for (int k = 0;k < n;k++)
    for (int j = 0;j < 2;j++)
      y[k][j] = false;
  for (int k = 0;k <= E.s;k++)
    for (int j = 0;j < 2;j++)
      y[a[E.x[k]][j]][j] = true;
```

```
f1 = E.f1 + M[E.x[E.s]][0];
f2 = ((f1 > E.f2)?f1:E.f2) + M[E.x[E.s]][1];
int sf2 = E.sf2 + f2;
int s1 = 0,s2 = 0,k1 = n - E.s,k2 = n - E.s,f3 = f2;
// 计算 s1 的值
for (int j = 0;j < n;j ++)
  if (!y[j][0]) {
    k1 --;
    if (k1 == n - E.s - 1)
      f3 = (f2 > f1 + b[j][0])?f2:f1 + b[j][0];
    s1 += f1 + k1 * b[j][0];}
    // 计算 s2 的值
    for (int j = 0;j < n;j ++)
      if (!y[j][1]) {
        k2 --;
        s1 += b[j][1];
        s2 += f3 + k2 * b[j][1];}
    // 返回完成时间和下界
    return sf2 + ((s1 > s2)?s1:s2);
  }
```

函数 BBFlow 是解批处理作业调度问题的优先队列式分支限界法的主体。算法开始时，将排列树的根结点置为当前扩展结点。在初始扩展结点处还没有选定的作业，故 $s = 0$，数组 x 初始化为单位排列。

算法的 while 循环完成对排列树内部结点的有序扩展。在 while 循环体内算法依次从活结点优先队列中取出具有最小 bb 值的结点作为当前扩展结点，并加以扩展。

算法将当前扩展结点 E 分为两种情形处理。首先考虑 E.$s = n$ 的情形，此时已排定 n 个作业，故当前扩展结点 E 是排列树中的一个叶结点。E.x 表示相应于该叶结点的作业调度。E.sf2 是相应于该叶结点的完成时间和。当 E.sf2 < bestc 时更新当前最优值 bestc 和相应的当前最优解 bestx。

当 E.$s < n$ 时，算法依次产生当前扩展结点 E 的所有儿子结点。对于当前扩展结点的每一个儿子结点 N，计算出其相应的完成时间和的下界 bb。当 bb < bestc 时，将该儿子结点 N 插入到活结点优先队列中。而当 bb ≥ bestc 时，以 N 为根的子树中不可能有比当前最优解 bestx 更好的解，故将结点 N 舍去。

解批处理作业调度问题的优先队列式分支限界法可描述如下：

```
int Flowshop::BBFlow(void)
{// 解批处理作业调度问题的优先队列式分支限界法
    Sort(); // 对各作业在机器 1 和 2 上所需时间排序
    // 定义最小堆的容量为 1000
    MinHeap < MinHeapNode > H(1000);
    MinHeapNode E;
```

```
        // 初始化
        E.Init(n);
        // 搜索排列空间树
        while (E.s <= n ) {
          if (E.s == n ) {// 叶结点
            if (E.sf2 < bestc) {
              bestc = E.sf2;
              for (int i = 0; i < n; i++)
                bestx[i] = E.x[i];}
            delete [] E.x;}
          else {// 产生当前扩展结点的儿子结点
          for (int i = E.s; i < n; i++) {
            Swap(E.x[E.s],E.x[i]);
            int f1,f2;
            int bb = Bound(E,f1,f2,y);
            if (bb < bestc ) {
                // 子树可能含最优解
                // 结点插入最小堆
                MinHeapNode N;
                N.NewNode(E.f1,f2,bb,n);
                H.Insert(N);}
            Swap(E.x[E.s],E.x[i]);
            }
      delete [] E.x;}  // 完成结点扩展
    try {H.DeleteMin(E);}         // 取下一扩展结点
    catch (OutOfBounds) {break;} // 堆已空
    }
  return bestc;
}
```

习题 6

6-1 栈式分支限界法将活结点表以后进先出(LIFO)的方式存储于一个栈中。试设计一个解 0-1 背包问题的栈式分支限界法，并说明栈式分支限界法与回溯法的区别。

6-2 修改解装载问题的分支限界算法 MaxLoading，使得算法在结束前释放所有已由 EnQueue 产生的结点。

6-3 解装载问题的分支限界算法中，由 EnQueue 产生的结点可以在算法结束前一次性删除。然而那些没有活儿子结点或没有叶结点的扩展结点可以立即被删除。试设计一个在算法中及时删除不用结点的方案，并讨论其时间与空间之间的折衷。

6-4 试修改解装载问题和解0-1 背包问题的优先队列式分支限界法，使其仅使用一个最大堆来存储活结点，而不必存储所产生的解空间树。

6-5　试修改解装载问题和解0-1背包问题的优先队列式分支限界法，使得算法在运行结束时释放所有类型为 bbnode 和 HeapNode 的结点所占用的空间。

6-6　在解最大团问题的优先队列式分支限界法中，当前扩展结点满足 $cn + n - i \geq bestn$ 的右儿子结点被插入到优先队列中。如果将这个条件修改为满足 $cn + n - i > bestn$ 右儿子结点插入优先队列，仍能保证算法的正确性吗？为什么？

6-7　考虑最大团问题的子集空间树中第 i 层的一个结点 x，设 MinDegree(x) 是以结点 x 为根的子树中所有结点度数的最小值。

(1) 设 $x.u = \min\{x.cn + n - i + 1, \text{MinDegree}(x) + 1\}$，证明以结点 x 为根的子树中任一叶结点所相应的团的大小不超过 $x.u$。

(2) 依此 $x.u$ 的定义重写算法 BBMaxClique。

(3) 比较新旧算法所需的计算时间和产生的排列树结点数。

6-8　试修改解旅行售货员问题的分支限界法，使得 $s = n - 2$ 的结点不插入优先队列，而是将当前最优排列存储于 bestp 中。经这样修改后，算法在下一个扩展结点满足条件 Lcost $\geq$ bestc 时结束。

6-9　试修改解旅行售货员问题的分支限界法，使得算法保存已产生的排列树。

6-10　试设计解电路板排列问题的队列式分支限界法，并使算法在运行结束时输出最优解和最优值。

6-11　试设计一个解最小长度电路板排列问题(见习题 5-9) 的队列式分支限界法。

6-12　用优先队列式分支限界法解最小长度电路板排列问题。

6-13　试设计一个解图的顶点覆盖问题的优先队列式分支限界法。

6-14　试设计一个解图的最大割点问题的优先队列式分支限界法。

6-15　试设计一个解最小重量机器设计问题(习题5-10) 的优先队列式分支限界法。

6-16　试设计一个解运动员最佳配对问题(习题5-14) 的优先队列式分支限界法。

6-17　试设计一个解网络设计问题(习题5-20) 的优先队列式分支限界法。

6-18　试设计一个解 n 后问题的优先队列式分支限界法。

6-19　试设计一个解圆排列问题的优先队列式分支限界法。

6-20　试设计一个解布线问题(习题5-25) 的优先队列式分支限界法。

6-21　试设计一个解最佳调度问题(习题5-26) 的优先队列式分支限界法。

6-22　试设计一个解无优先级运算问题(习题5-27) 的优先队列式分支限界法。

6-23　试设计一个解最大 k 乘积问题(习题5-28) 的优先队列式分支限界法。

6-24　试设计一个解世界名画陈列馆问题(习题5-29) 的优先队列式分支限界法。

6-25　用队列式分支限界法重做习题5-11。

6-26　用队列式分支限界法重做习题5-12。

6-27　用队列式分支限界法重做习题5-13。

6-28　用优先队列式分支限界法重做习题5-11。

6-29　用优先队列式分支限界法重做习题5-12。

6-30　用优先队列式分支限界法重做习题5-13。

第7章 概率算法

学习要点

- 理解产生伪随机数的算法
- 掌握数值概率算法的设计思想
- 掌握蒙特卡罗算法的设计思想
- 掌握拉斯维加斯算法的设计思想
- 掌握舍伍德算法的设计思想

前面各章中所讨论算法的每一计算步骤都是确定的，而本章所讨论的概率算法允许算法在执行过程中随机地选择下一个计算步骤。在许多情况下，当算法在执行过程中面临一个选择时，随机性选择常比最优选择省时。因此概率算法可在很大程度上降低算法的复杂度。

概率算法的一个基本特征是对所求解问题的同一实例用同一概率算法求解两次可能得到完全不同的效果。这两次求解所需的时间甚至所得到的结果可能会有相当大的差别。一般情况下，可将概率算法大致分为四类：数值概率算法、蒙特卡罗（Monte Carlo）算法、拉斯维加斯（Las Vegas）算法和舍伍德（Sherwood）算法。

数值概率算法常用于数值问题的求解。这类算法所得到的往往是近似解。且近似解的精度随计算时间的增加而不断提高。在许多情况下，要计算出问题的精确解是不可能的或没有必要的，因此用数值概率算法可得到相当满意的解。

蒙特卡罗方法用于求问题的准确解。对于许多问题来说，近似解毫无意义。例如，一个判定问题其解为“是”或“否”，二者必居其一，不存在任何近似解答。又如，我们要求一个整数的因子时所给出的解答必须是准确的，一个整数的近似因子没有任何意义。用蒙特卡罗算法能求得问题的一个解，但这个解未必是正确的。求得正确解的概率依赖于算法所用的时间。算法所用的时间越多，得到正确解的概率就越高。蒙特卡罗算法的主要缺点也在于此。一般情况下，无法有效地判定所得到的解是否肯定正确。

拉斯维加斯算法不会得到不正确的解。一旦用拉斯维加斯算法找到一个解，这个解就一定是正确解。但有时用拉斯维加斯算法会找不到解。与蒙特卡罗算法类似，拉斯维加斯算法找到正确解的概率随着它所用的计算时间的增加而提高。对于所求解问题的任一实例，用同一拉斯维加斯算法反复对该实例求解足够多次，可使求解失效的概率任意小。

舍伍德算法总能求得问题的一个解，且所求得的解总是正确的。当一个确定性算法在最坏情况下的计算复杂性与其在平均情况下的计算复杂性有较大差别时，可在这个确定性算法中引入随机性将它改造成一个舍伍德算法，消除或减少问题的好坏实例间的这种差别。舍伍德算法精髓不是避免算法的最坏情况行为，而是设法消除这种最坏情形行为与特定实例之间的关联性。

在本章的后续各节中将分别讨论上述4类概率算法。

7.1 随机数

随机数在概率算法设计中扮演着十分重要的角色。在现实计算机上无法产生真正的随机数，因此在概率算法中使用的随机数都是一定程度上随机的，即伪随机数。

产生伪随机数最常用的方法是线性同余法。由线性同余法产生的随机序列 $a_1, a_2, \cdots, a_n, \cdots$ 满足

$$\begin{cases} a_0 = d \\ a_n = (ba_{n-1} + c) \bmod m \qquad n = 1, 2, \cdots \end{cases}$$

其中，$b \geqslant 0$，$c \geqslant 0$，$d \geqslant m$。d 称为该随机序列的种子。如何选取该方法中的常数 b，c 和 m 直接关系到所产生的随机序列的随机性能。这是随机性理论研究的内容，已超出本书讨论的范围。从直观上看，m 应取得充分大，因此可取 m 为机器大数 65 536，另外应取 $\gcd(m, b) = 1$，因此可取 b 为一素数。

为了在设计概率算法时便于产生所需的随机数，建立一个随机数类 RandomNumber。该类包含一个需由用户初始化的种子 randSeed。给定初始种子后，即可产生与之相应的随机序列。种子 randSeed 是一个无符号长整型数，可由用户选定也可用系统时间自动产生。函数 Random 的输入参数 $n \leqslant 65\ 536$ 是一个无符号长整型数，它返回 $0 \sim (n-1)$ 范围内的一个随机整数。函数 fRandom 返回 $[0,1)$ 内的一个随机实数。

```
// 随机数类
const unsigned long maxshort = 65536L;
const unsigned long multiplier = 1194211693L;
const unsigned long adder = 12345L;

class RandomNumber
{
    private:
        // 当前种子
        unsigned long randSeed;
    public:
        // 构造函数，缺省值 0 表示由系统自动产生种子
        RandomNumber(unsigned long s = 0);
        // 产生 0:n-1 之间的随机整数
        unsigned short Random(unsigned long n);
        // 产生[0,1)之间的随机实数
        double fRandom(void);
};
```

函数 Random 在每次计算时，用线性同余式计算新的种子 randSeed。它的高 16 位的随机性较好。将 randSeed 右移 16 位得到一个 0～65 535 间的随机整数，然后再将此随机整数映射到 $0 \sim (n-1)$ 范围内。

对于函数 fRandom，我们先用函数 Random(maxshort) 产生一个 0～(maxshort－1) 之间的

整型随机序列,将每个整型随机数除以 maxshort,就得到[0,1)区间中的随机实数。

```
// 产生种子
RandomNumber::RandomNumber (unsigned long s)
{
    if (s == 0)
        randSeed = time(0);  // 用系统时间产生种子
    else
        randSeed = s;        // 由用户提供种子
}

// 产生 0:n-1 之间的随机整数
unsigned short RandomNumber::Random (unsigned long n)
{
    randSeed = multiplier * randSeed + adder;
    return (unsigned short)((randSeed >> 16)  % n);
}

// 产生[0,1)之间的随机实数
double RandomNumber::fRandom (void)
{
    return Random(maxshort)/double(maxshort);
}
```

下面用计算机产生的伪随机数来模拟抛硬币试验。假设抛 10 次硬币,每次抛硬币得到正面和反面是随机的。抛 10 次硬币构成一个事件。函数调用 Random(2)返回一个二值结果。返回 0 表示抛硬币得到反面,返回 1 表示得到正面。下面的函数 TossCoins 模拟抛 10 次硬币这一事件。在主程序中反复用函数 TossCoins 模拟抛 10 次硬币这一事件 50 000 次。用 head[i]($0 \leqslant i \leqslant 10$)记录这 50 000 次模拟恰好得到 i 次正面的次数。主程序最终输出模拟抛硬币事件得到正面事件的频率图,如图 7-1 所示。

```
int TossCoins(int numberCoins)
{ // 随机抛硬币
    static RandomNumber coinToss;
    int i, tosses = 0;
    for (i=0;i < numberCoins;i++)
        // Random(2) = 1 表示正面
        tosses += coinToss.Random(2);
    return tosses;
}

void main(void)
{ // 模拟随机抛硬币事件
    const int NCOINS = 10;
    const long NTOSSES = 50000L;
// heads[i]是得到 i 次正面的次数
    long i, heads[NCOINS + 1];
```

```
    int  j, position;
// 初始化数组 heads
    for (j=0;j < NCOINS+1;j++)
        heads[j] = 0;
// 重复 50,000 次模拟事件
    for (i=0;i < NTOSSES;i++)
        heads[TossCoins(NCOINS)]++;
// 输出频率图
    for (i=0; i <= NCOINS;i++)
    {
        position = int(float(heads[i])/NTOSSES * 72);
        cout << setw(6) << i << " ";
        for (j = 0; j < position-1;j++)
            cout << " ";
        cout << "*" << endl;
    }
}
```

```
 0  *
 1  *
 2     *
 3          *
 4                *
 5                    *
 6                *
 7          *
 8     *
 9  *
10  *
```

图 7-1 模拟抛硬币得到的正面事件频率图

7.2 数值概率算法

7.2.1 用随机投点法计算 π 值

设有一半径为 r 的圆及其外切四边形,如图 7-2(a)所示。向该正方形随机地投掷 n 个点。设落入圆内的点数为 k。由于所投入的点在正方形上均匀分布,因而所投入的点落入圆内的概率为$\frac{\pi r^2}{4r^2}=\frac{\pi}{4}$,所以当 n 足够大时,k 与 n 之比就逼近这一概率,即$\frac{\pi}{4}$。从而 $\pi\approx\frac{4k}{n}$。由此可得用随机投点法计算 π 值的数值概率算法。在具体实现时,只要在第一象限计算即可,如图 7-2(b)所示。

```
double Darts(int n)
{ // 用随机投点法计算 π 值
    static RandomNumber dart;
    int k = 0;
    for (int i = 1;i < = n;i + + ) {
      double x = dart.fRandom( );
      double y = dart.fRandom( );
      if ((x * x + y * y) < = 1) k + + ;
      }
    return 4 * k/double(n);
}
```

7.2.2 计算定积分

(1) 用随机投点法计算定积分

设 $f(x)$ 是[0,1] 上的连续函数,且 $0 \leqslant f(x) \leqslant 1$。需要计算积分值 $I = \int_0^1 f(x)\mathrm{d}x$。积分 I 等于图 7-3 中的面积 G。

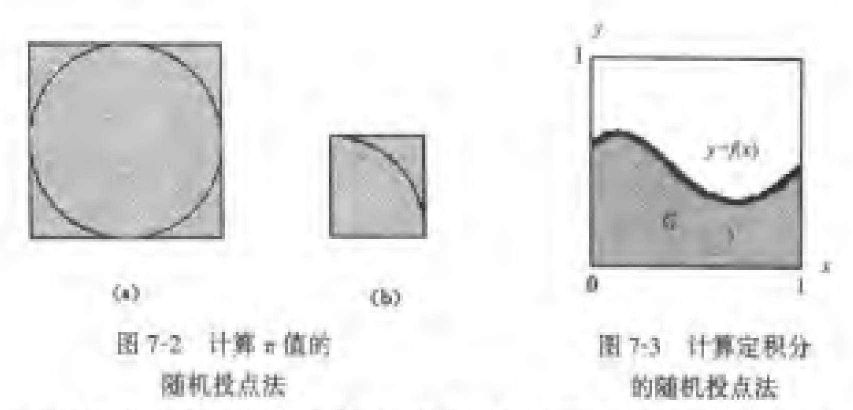


图 7-2 计算 π 值的随机投点法

图 7-3 计算定积分的随机投点法

在图 7-3 所示单位正方形内均匀地作投点试验,则随机点落在曲线 $y = f(x)$ 下面的概率为

$$P_r\{y \leqslant f(x)\} = \int_0^1\int_0^{f(x)} \mathrm{d}y\mathrm{d}x = \int_0^1 f(x)\mathrm{d}x = I$$

假设向单位正方形内随机地投入 n 个点(x_i,y_i),$i = 1,\cdots,n$。随机点(x_i,y_i)落入 G 内,则 $y_i \leqslant f(x_i)$。如果有 m 个点落入 G 内,则 $\bar{I} = \frac{m}{n}$ 近似等于随机点落入 G 内的概率,即 $I \approx \frac{m}{n}$。

由此可设计出计算积分 I 的数值概率算法。

```
double Darts(int n)
{ // 用随机投点法计算定积分
    static RandomNumber dart;
```

```
    int k = 0;
    for (int i = 1; i <= n; i++) {
        double x = dart.fRandom();
        double y = dart.fRandom();
        if (y <= f(x)) k++;
    }
    return k/double(n);
}
```

如果所遇到的积分形式为 $I = \int_a^b f(x)\mathrm{d}x$，其中，$a$ 和 b 为有限值；被积函数 $f(x)$ 在区间 $[a,b]$ 中有界，并用 M，L 分别表示其最大值和最小值。此时可作变量代换 $x = a + (b-a)z$，将所求积分变为 $I = cI^* + d$。其中，

$$c = (M - L)(b - a),\ d = L(b - a),\ I^* = \int_0^1 f^*(z)\mathrm{d}z$$

$$f^*(z) = \frac{1}{M-L}[f(a + (b-a)z) - L] \quad (0 \leqslant f^*(z) \leqslant 1)$$

因此，I^* 可用随机投点法计算。

（2）用平均值法计算定积分

任取一组相互独立、同分布的随机变量 $\{\xi_i\}$，ξ_i 在 $[a,b]$ 中服从分布律 $f(x)$，令 $g^*(x) = \frac{g(x)}{f(x)}$，则 $\{g^*(\xi_i)\}$ 也是一组互相独立、同分布的随机变量，而且

$$E(g^*(\xi_i)) = \int_a^b g^*(x)f(x)\mathrm{d}x = \int_a^b g(x)\mathrm{d}x = I$$

由强大数定理

$$P_r\left(\lim_{n\to\infty}\frac{1}{n}\sum_{i=1}^{n} g_*(\xi_i) = I\right) = 1$$

若选 $\overline{I} = \frac{1}{n}\sum_{i=1}^{n} g^*(\xi_i)$，则 $\overline{I}$ 依概率 1 收敛于 I。平均值法就是用 $\overline{I}$ 作为 I 的近似值。

假设要计算的积分形式为 $I = \int_a^b g(x)\mathrm{d}x$，其中被积函数 $g(x)$ 在区间 $[a,b]$ 内可积。任意选择一个有简便方法可以进行抽样的概率密度函数 $f(x)$，使其满足下列条件：

① $f(x) \neq 0$，当 $g(x) \neq 0$ 时（$a \leqslant x \leqslant b$）；

② $\int_a^b f(x)\mathrm{d}x = 1$。

如果记

$$g^*(x) = \begin{cases} \frac{g(x)}{f(x)} & f(x) \neq 0 \\ 0 & f(x) = 0 \end{cases}$$

则所求积分可以写为

$$I = \int_a^b g^*(x) f(x) \mathrm{d}x$$

由于 a 和 b 为有限值,可取 $f(x)$ 为均匀分布:

$$f(x) = \begin{cases} \frac{1}{b-a} & a \leqslant x \leqslant b \\ 0 & x < a, x > b \end{cases}$$

这时所求积分变为

$$I = (b-a)\int_a^b g(x) \frac{1}{b-a} \mathrm{d}x$$

在$[a,b]$区间上随机抽取个点 $x_i,(i = 1,2,\cdots,n)$,则均值$\overline{I} = \frac{b-a}{n}\sum_{i=1}^{n} g(x_i)$可作为所求积分 I 的近似值。

由此可设计出计算积分 I 的平均值法如下:

```
double Integration(double a, double b, int n)
{ // 用平均值法计算定积分
    static RandomNumber rnd;
    double y = 0;
    for (int i = 1;i < = n;i ++) {
      double x = (b - a) * rnd.fRandom() + a;
      y + = g(x);
      }
    return (b - a) * y/double(n);
}
```

7.2.3 解非线性方程组

假设我们要求解下面的非线性方程组

$$\begin{cases} f_1(x_1,x_2,\cdots,x_n) = 0 \\ f_2(x_1,x_2,\cdots,x_n) = 0 \\ \cdots \\ f_n(x_1,x_2,\cdots,x_n) = 0 \end{cases}$$

其中,$x_1,x_2,\cdots,x_n$ 是实变量,$f_i(i = 1,2,\cdots,n)$ 是未知量 $x_1,x_2,\cdots,x_n$ 的非线性实函数。我们要求上述方程组在指定求根范围内的一组解 $x_1^*,x_2^*,\cdots,x_n^*$。

解决这类问题有许多种数值方法。最常用的有线性化方法和求函数极小值方法。应当指出,在使用某种具体算法求解的过程中,有时会遇到一些麻烦,甚至于使方法失效而不能获得一个近似解。在这种情况下,我们可以求助于概率算法. 一般而言,概率算法需耗费较多时间,但其设计思想简单,易于实现,因此在实际使用中还是比较有效的。对于精度要求较高的问题,概率算法常常可以提供一个较好的初值。下面介绍求解非线性方程组的概率算法的基本思想。

为了求解所给的非线性方程组,构造一目标函数

$$\Phi(x) = \sum f_i^2(x)$$

其中，$x = (x_1, x_2, \cdots, x_n)$。由最优化理论可知，该目标函数 $\Phi(x)$ 的极小值点即是所求非线性方程组的一组解。

在求函数 $\Phi(x)$ 的极小值点时可采用简单随机模拟算法，在指定求根区域内，选定一个 x_0 作为根的初值。按照预先选定的分布(如以 x_0 为中心的正态分布，均匀分布，三角分布等)，逐个选取随机点 x，计算目标函数 $\Phi(x)$，并把满足精度要求的随机点 x 作为所求非线性方程组的近似解。这种方法直观、简单，但工作量较大。下面介绍的随机搜索算法可以克服这一缺点。

在指定求根区域 D 内，选定一个随机点 x_0 作为随机搜索的出发点。在搜索过程中，假设第 j 步随机搜索得到的随机搜索点为 x_j。在第 $j+1$ 步，首先计算出下一步的随机搜索方向 r；然后计算搜索步长 a。由此得到第 $j+1$ 步的随机搜索增量 Δx_j。从当前点 x_j 依随机搜索增量 Δx_j 得到第 $j+1$ 步的随机搜索点 $x_{j+1} = x_j + \Delta x_j$。当 $\Phi(x_{j+1}) < \varepsilon$ 时，取 x_{j+1} 为所求非线性方程组的近似解。否则进行下一步新的随机搜索过程。

具体算法可描述如下：

```
bool NonLinear(double * x0, double * dx0, double * x, double a0,
            double epsilon, double k, int n, int Steps, int M)
{ // 解非线性方程组的概率算法
      static RandomNumber rnd;
      bool success;                    // 搜索成功标志
      double * dx, * r;
      dx = new double [n + 1];         // 步进增量向量
      r = new double [n + 1];          // 搜索方向向量
      int mm = 0;                      // 当前搜索失败次数
      int j = 0;         // 迭代次数
      double a = a0;  // 步长因子
      for (int i = 1;i < = n;i ++) {
        x[i] = x0[i];
        dx[i] = dx0[i];
        }
      double fx = f(x,n);   // 计算目标函数值
      double min = fx;       // 当前最优值
      while ((min > epsilon) && (j < Steps)) {
        // (1) 计算随机搜索步长
        if (fx < min) { // 搜索成功
          min = fx;
          a* = k;
          success = true;}
        else { // 搜索失败
          mm ++;
          if (mm > M) a/ = k;
          success = false;}
        // (2) 计算随机搜索方向和增量
        for (int i = 1;i < = n;i ++)
          r[i] = 2.0 * rnd.fRandom() - 1;
```

```
        if (success)
          for (int i = 1;i <= n;i ++)
            dx[i] = a * r[i];
        else
          for (int i = 1;i <= n;i ++)
            dx[i] = a * r[i] - dx[i];
        // (3) 计算随机搜索点
        for (int i = 1;i <= n;i ++)
          x[i] += dx[i];
        // (4) 计算目标函数值
        fx = f(x,n);
      }
      if (fx <= epsilon) return true;
      else return false;
}
```

7.3 舍伍德(Sherwood)算法

我们分析一个算法在平均情况下的计算复杂性时，通常假定算法的输入数据服从某一特定的概率分布。例如，在输入数据是均匀分布时，快速排序算法所需的平均时间是 $O(n\ \log n)$。而当其输入已“几乎”排好序时，这个时间界就不再成立。此时，可采用舍伍德算法消除算法所需计算时间与输入实例间的这种联系。

设 A 是一个确定性算法，当它的输入实例为 x 时所需的计算时间记为 $t_A(x)$。设 X_n 是算法 A 的输入规模为 n 的实例的全体，则当问题的输入规模为 n 时，算法 A 所需的平均时间为

$$\bar{t}_A(n) = \sum_{x \in X_n} t_A(x) / |X_n|$$

这显然不能排除存在 $x \in X_n$ 使得 $t_A(x) >> \bar{t}_A(n)$ 的可能性。我们希望获得一个概率算法 B，使得对问题的输入规模为 n 的每一个实例 $x \in X_n$ 均有 $t_B(x) = t_A(n) + s(n)$。对于某一具体实例 $x \in X_n$，算法 B 偶尔需要较 $\bar{t}_A(n) + s(n)$ 多的计算时间。但这仅仅是由于算法所作的概率选择引起的，与具体实例 x 无关。我们定义算法 B 关于规模为 n 的随机实例的平均时间为

$$t_B(n) = \sum_{x \in X_n} t_B(x) / (X_n)$$

易知 $t_B(n) = t_A(n) + s(n)$。这就是舍伍德算法设计的基本思想。当 $s(n)$ 与 $t_A(n)$ 相比可忽略时，舍伍德算法可获得很好的平均性能。

7.3.1 线性时间选择算法

在第 2 章中我们讨论了快速排序算法和线性时间选择算法。这两个算法的随机化版本就是舍伍德型概率算法。这两个算法的核心都在于选择合适的划分基准。对于选择问题而言，用拟中位数作为划分基准可以保证在最坏情况下用线性时间完成选择。如果只简单地用待划分数组的第一个元素作为划分基准，则算法的平均性能较好，而在最坏情况下需要 $O(n^2)$ 计算时间。舍伍德型选择算法则随机地选择一个数组元素作为划分基准，这样既能保证算法的线性

时间平均性能又避免了计算拟中位数的麻烦。

非递归的舍伍德型选择算法可描述如下：

```
template < class Type >
Type select(Type a[], int l, int r, int k)
{// 计算 a[l:r] 中第 k 小元素
  static RandomNumber rnd;
  while (true) {
   if (l > = r) return a[l];
   int i = l,
       j = l + rnd.Random(r - l + 1);  // 随机选择的划分基准
   Swap(a[i], a[j]);
   j = r + 1;
   Type pivot = a[l];
   // 以划分基准为轴作元素交换
   while (true) {
          while (a[ + + i] < pivot);
          while (a[ - - j] > pivot);
      if (i > = j) break;
      Swap(a[i], a[j]);
      }
   if (j - l + 1 = = k) return pivot;
   a[l] = a[j];
   a[j] = pivot;
   // 对子数组重复划分过程
   if (j - l + 1 < k) {
     k = k - j + l - 1;
     l = j + 1;}
   else r = j - 1;
  }
}

template < class Type >
Type Select(Type a[], int n, int k)
{// 计算 a[0:n - 1] 中第 k 小元素
 // 假设 a[n] 是一个键值无穷大的元素
   if (k < 1 || k > n) throw OutOfBounds();
   return select(a, 0, n - 1, k);
}
```

由于算法 select 使用了一个随机数产生器随机地产生 l 和 r 之间的一个随机整数，因此，算法 select 所产生的划分基准是随机的。可以证明，当用算法 select 对含有 n 个元素的数组进行划分时，划分出的低区子数组中含有一个元素的概率为 $2/n$；含有 i 个元素的概率为 $1/n$，$i = 2,3,\cdots,n-1$。今设 $T(n)$ 是算法 select 作用于一个含有 n 个元素的输入数组上所需的期望时间的一个上界，且 $T(n)$ 是单调递增的。在最坏情况下，第 k 小元素总是被划分在较大的子数组中。由此，我们可以得到关于 $T(n)$ 的递归式：

$$T(n) \leqslant \frac{1}{n}\Big(T(\max(1, n-1)) + \sum_{i=1}^{n-1} T(\max(i, n-i))\Big) + O(n)$$
$$\leqslant \frac{1}{n}\Big(T(n-1) + 2\sum_{i=n/2}^{n-1} T(i)\Big) + O(n)$$
$$= \frac{2}{n}\sum_{i=n/2}^{n-1} T(i) + O(n)$$

在上面的推导中，从第1行到第2行是因为 $\max(1, n-1) = n-1$，而

$$\max(i, n-i) = \begin{cases} i & i \geqslant n/2 \\ n-i & i < n/2 \end{cases}$$

且 n 是奇数时，$T(n/2)$，$T(n/2+1)$，$\cdots$，$T(n-1)$ 在和式中均出现2次；n 是偶数时，$T(n/2+1)$，$T(n/2+2)$，$\cdots$，$T(n-1)$ 均出现2次，$T(n/2)$ 只出现1次。因此，第2行中的和式是第1行中和式的一个上界。从第2行到第3行是因为在最坏情况下 $T(n-1) = O(n^2)$，故可将 $T(n-1)/n$ 包含在 $O(n)$ 项中。

解上面的递归式可得 $T(n) = O(n)$。换句话说，非递归的舍伍德型选择算法 select 可以在 $O(n)$ 平均时间内找出 n 个输入元素中的第 k 小元素。

综上所述，我们开始时所考虑的是一个有很好平均性能的选择算法，但在最坏情况下对某些实例算法效率较低。在这种情况下，我们采用概率方法，将上述算法改造成一个舍伍德型算法，使得该算法以高概率对任何实例均有效。对于舍伍德型快速排序算法，分析是类似的。

上述舍伍德型选择算法对确定性选择算法所作的修改非常简单且容易实现。但有时所给的确定性算法无法直接改造成舍伍德型算法。此时可借助于随机预处理技术，不改变原有的确定性算法，仅对其输入进行随机洗牌，同样可收到舍伍德算法的效果。例如，对于确定性选择算法，我们可以用下面的洗牌算法 Shuffle 将数组 a 中元素随机排列，然后用确定性选择算法求解。这样做所收到的效果与舍伍德型算法的效果是一样的。

```
template < class Type >
void Shuffle(Type a[], int n)
{// 随机洗牌算法
  static RandomNumber rnd;
  for (int i = 0;i < n;i ++) {
    int j = rnd.Random(n - i) + i;
    Swap(a[i], a[j]);
    }
}
```

7.3.2 搜索有序表

有序字典是表示有序集很有用的抽象数据类型。它支持对有序集的搜索、插入、删除、前驱、后继等运算。有许多基本数据结构可用于实现有序字典。下面我们讨论其中的一种基本数据结构。

我们用两个数组来表示所给的含有 n 个元素的有序集 S。用 value[0: n] 存储有序集中的元素，link[0: n] 存储表示元素在数组 value 中位置的指针。link[0] 指向有序集中第1个元素。换句话说，value[link[0]] 是集合中的最小元素。一般地，如果 value[i] 是所给有序集 S 中的第 k 个元素，

则 value[link[i]] 是 S 中的第 $k+1$ 个元素。S 中元素的有序性表现为，对于任意 $1 \leq i \leq n$ 有 value[i] $\leq$ value[link[i]]。集合 S 中的最大元素 value[k] 有，link[k] = 0 且 value[0] 是一个大数。例如，表示有序集 $S = \{1,2,3,5,8,13,21\}$ 的一种表示方式如图 7-4 所示。

i	0	1	2	3	4	5	6	7
value[i]	∞	2	3	13	1	5	21	8
link[i]	4	2	5	6	1	7	0	3

图 7-4　用数组表示有序集

在此例中，link[0] = 4 指向 S 中最小元素 value[4] = 1。显而易见，这种表示有序集的方法实际上是用数组来模拟有序链表。对于有序链表，可采用顺序搜索的方式在所给的有序集 S 中搜索链值为 x 的元素。如果有序集 S 中含有 n 个元素，则在最坏情况下，顺序搜索算法所需的计算时间为 $O(n)$。

利用数组下标的索引性质，我们可以设计一个随机化的搜索算法，以改进算法的搜索时间复杂性。算法的基本思想是随机抽取数组元素若干次，从较接近搜索元素 x 的位置开始作顺序搜索。可以证明，如果随机抽取数组元素 k 次，则其后顺序搜索所需的平均比较次数为 $O(n/(k+1))$。因此如果取 $k = \lfloor \sqrt{n} \rfloor$，则算法所需的平均计算时间为 $O(\sqrt{n})$。

下面讨论上述算法的实现细节。用数组来表示的有序链表由类 OrderedList 定义如下：

```
template < class Type >
class OrderedList {
  public:
      OrderedList(Type small, Type Large, int MaxL);
       ~ OrderedList();
      bool Search(Type x, int& index);    // 搜索指定元素
      int SearchLast(void);               // 搜索最大元素
      void Insert(Type k);                // 插入指定元素
      void Delete(Type k);                // 删除指定元素
      void Output();                      // 输出集合中元素
  private:
      int n;                   // 当前集合中元素个数
      int MaxLength;           // 集合中最大元素个数
      Type * value;            // 存储集合中元素的数组
      int * link;              // 指针数组
      RandomNumber rnd;        // 随机数产生器
      Type Small;              // 集合中元素的下界
      Type TailKey;            // 集合中元素的上界
};

template < class Type >
OrderedList < Type > ::OrderedList(Type small, Type Large, int MaxL)
{// 构造函数
   MaxLength = MaxL;
   value = new Type [MaxLength + 1];
   link = new int [MaxLength + 1];
   TailKey = Large;
```

```
    n = 0;
    link[0] = 0;
    value[0] = TailKey;
    Small = small;
}
```

```
template < class Type >
OrderedList < Type > :: ~ OrderedList()
{// 析构函数
    delete value;
    delete link;
}
```

其中,MaxLength 是集合中元素个数的上限;Small 和 TailKey 分别是全集合中元素的下界和上界;OrderedList 的构造函数初始化其私有成员数组 value 和 link,它的析构函数则释放 value 和 link 占用的所有空间。

OrderedList 类的共享成员函数 Search 用来搜索当前集合中的元素 x。当 Search 搜索到元素 x 时,将该元素在数组 value 中的位置返回到 index 中,并返回 true,否则返回 false。

```
template < class Type >
bool OrderedList < Type > ::Search(Type x, int& index)
{// 搜索集合中指定元素 k
    index = 0;
    Type max = Small;
    int m = floor(sqrt(double(n)));  // 随机抽取数组元素次数
    for (int i = 1; i <= m; i++) {
     int j = rnd.Random(n) + 1;      // 随机产生数组元素位置
     Type y = value[j];
     if ((max < y) && (y < x)) {
        max = y;
        index = j;}
     }
   // 顺序搜索
   while (value[link[index]] < x) index = link[index];
   return (value[link[index]] == x);
}
```

有了函数 Search,就容易设计支持集合的插入和删除运算的算法 Insert 和 Delete 如下。插入运算首先用函数 Search 确认待插入元素 k 不在当前集合中,然后将新插入的元素存储在 value[$n+1$] 中,并修改相应的指针。Insert 所需的平均计算时间显然为 $O(\sqrt{n})$。

```
template < class Type >
void OrderedList < Type > ::Insert(Type k)
{// 插入指定元素
    if ((n == MaxLength) || (k >= TailKey)) return;
```

```
    int index;
    if (!Search(k,index)) {
    value[++n] = k;
    link[n] = link[index];
    link[index] = n;}
}
```

删除运算首先用函数 Search 找到待删除元素 k 在当前集合中的位置,然后修改待删除元素 k 的前驱元素的 link 指针,使其指向待删除元素 k 的后继元素。被删除元素 k 在有序表中产生的空洞,由当前集合中的最大元素来填补。搜索当前集合中的最大元素的任务由函数 SearchLast 来完成。与函数 Search 类似,函数 SearchLast 所需的平均计算时间也是 $O(\sqrt{n})$。因此,实现删除运算的算法 Delete 所需的平均计算时间为 $O(\sqrt{n})$。

```
template < class Type >
int OrderedList < Type > ::SearchLast(void)
{// 搜索集合中最大元素
   int index = 0;
   Type x = value[n];
   Type max = Small;
   int m = floor(sqrt(double(n)));   // 随机抽取数组元素次数
   for (int i = 1; i <= m; i++) {
     int j = rnd.Random(n) + 1;      // 随机产生数组元素位置
     Type y = value[j];
     if ((max < y) && (y < x)) {
       max = y;
       index = j;}
     }
   // 顺序搜索
   while (link[index]! = n) index = link[index];
   return index;
}
```

```
template < class Type >
void OrderedList < Type > ::Delete(Type k)
{// 删除集合中指定元素 k
   if ((n == 0) || (k >= TailKey)) return;
   int index;
   if (Search(k,index)) {
     int p = link[index];
     if (p == n) link[index] = link[p];
     else {
       if (link[p]! = n) {
         int q = SearchLast();
         link[q] = p;
         link[index] = link[p];}
```

```
            value[p] = value[n];
            link[p] = link[n];
            }
        n--;
        }
    }
```

7.3.3 跳跃表

舍伍德型算法的设计思想还可用于设计高效的数据结构，跳跃表就是一例。我们知道，如果用有序链表表示一个含有 n 个元素的有序集 S，则在最坏情况下，搜索 S 中一个元素需要 $\Omega(n)$ 计算时间。提高有序链表效率的一个技巧是在有序链表的部分结点处增设附加指针以提高其搜索性能。在增设附加指针的有序链表中搜索一个元素时，可借助于附加指针跳过链表中若干结点，加快搜索速度。这种增加了向前附加指针的有序链表称为跳跃表。应在跳跃表的哪些结点增加附加指针以及在该结点处应增加多少指针完全采用随机化方法来确定。这使得跳跃表可在 $O(\log n)$ 平均时间内支持关于有序集的搜索、插入和删除等运算。例如，图 7-5(a) 是一个没有附加指针的有序链表，而图 7-5(b) 在图 7-5(a) 的基础上增加了跳跃一个结点的附加指针，图 7-5(c) 在图 7-5(b) 的基础上又增加了跳跃 3 个结点的附加指针。

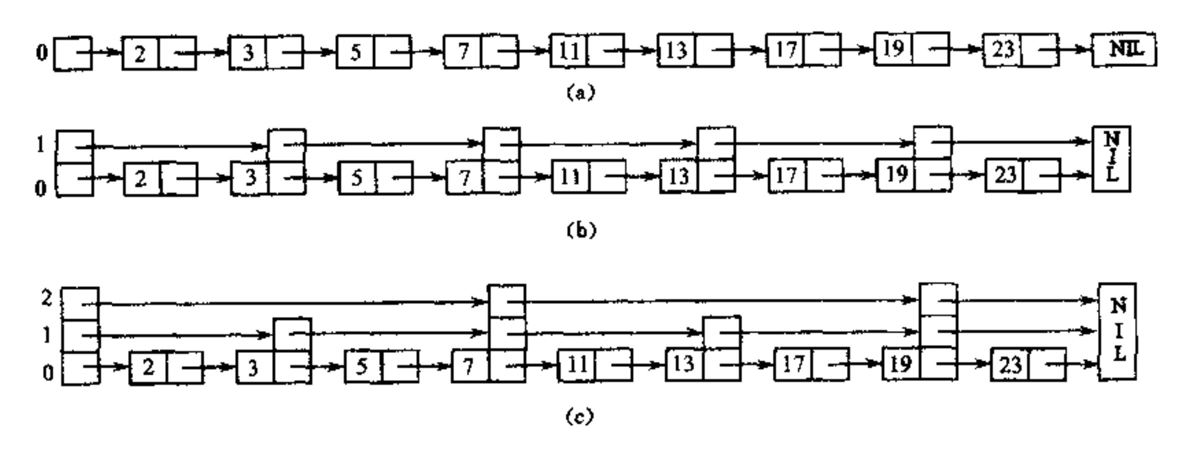


图 7-5　完全跳跃表

在跳跃表中，如果一个结点有 $k+1$ 个指针，则称此结点为一个 k 级结点。

以图 7-5(c) 中跳跃表为例，我们来看如何在该跳跃表中搜索元素 8。从该跳跃表的最高级，即第 2 级开始搜索。利用 2 级指针我们发现元素 8 位于结点 7 和 19 之间。此时在结点 7 处降至 1 级指针继续搜索，发现元素 8 位于结点 7 和 13 之间。最后，在结点 7 处降至 0 级指针进行搜索，发现元素 8 位于结点 7 和 11 之间，从而知道元素 8 不在所搜索的集合 S 中。

在一般情况下，给定一个含有 n 个元素的有序链表，我们可以将它改造成一个完全跳跃表，使得每一个 k 级结点含有 $k+1$ 个指针，分别跳过 $2^k-1, 2^{k-1}-1, \cdots, 2^0-1$ 个中间结点。第 i 个 k 级结点安排在跳跃表的位置 $i2^k$ 处，$i \geqslant 0$。这样就可以在 $O(\log n)$ 时间内完成集合成员的搜索运算。在一个完全跳跃表中，最高级的结点是 $\lceil \log n \rceil$ 结点。

完全跳跃表与完全二叉搜索树的情形非常类似。它虽然可以有效地支持成员搜索运算，但

不适应于集合动态变化的情况。集合元素的插入和删除运算会破坏完全跳跃表原有的平衡状态，影响后继元素搜索的效率。

为了在动态变化中维持跳跃表中附加指针的平衡性，必须使跳跃表中 k 级结点数维持在总结点数的一定比例范围内。注意到在一个完全跳跃表中，50% 的指针是 0 级指针；25% 的指针是 1 级指针；…；$(100/2^{k+1})\%$ 的指针是 k 级指针。因此，在插入一个元素时，我们以概率 1/2 引入一个 0 级结点，以概率 1/4 引入一个 1 级结点，…，以概率 $1/2^{k+1}$ 引入一个 k 级结点。另一方面，一个 i 级结点指向下一个同级或更高级的结点，它所跳过的结点数不再准确地维持在 2^i-1。经过这样的修改，我们就可以在插入或删除一个元素时，通过对跳跃表的局部修改来维持其平衡性。跳跃表中结点的级别在插入时确定，一旦确定便不再更改。图 7-6 是遵循上述原则的跳跃表的例子。对其进行搜索与对完全跳跃表所作的搜索是一样的。

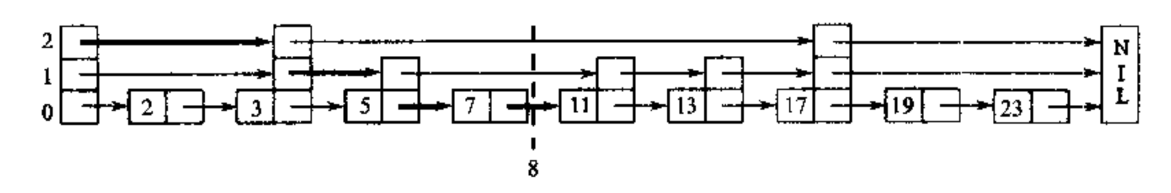


图 7-6　跳跃表示例

如果希望在图 7-6 所示的跳跃表中插入一个元素 8，则应先在跳跃表中搜索其插入位置。经搜索发现应在结点 7 和 11 之间插入元素 8。此时在结点 7 和 11 之间增加 1 个存储元素 8 的新结点，并以随机的方式确定新结点的级别。例如，如果元素 8 是作为一个 2 级结点插入，则应对图 7-6 中与虚线相交的指针进行调整如图7-7(a) 所示。如果新插入的结点是一个 1 级结点，则只要修改2个指针，如图 7-7(b) 所示。图 7-6中与虚线相交的指针是在插入新结点后有可能被修改的指针，这些指针可在搜索元素插入位置时动态地保存起来，以供实施插入时使用。

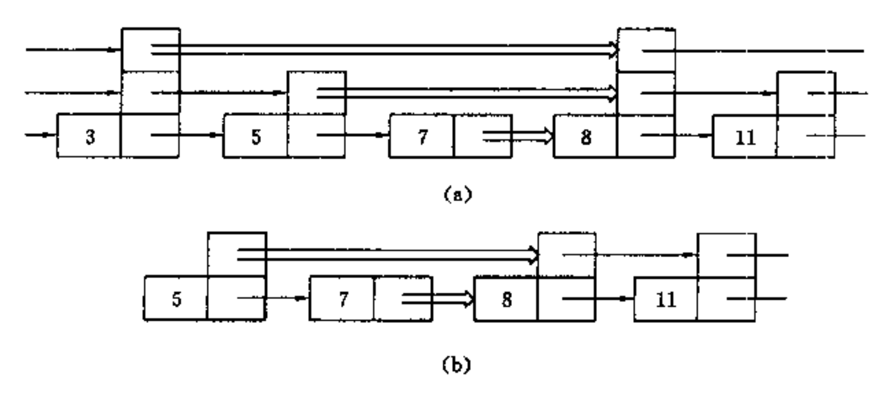


图 7-7　在跳跃表中插入新结点

在上述算法中，关键的问题是如何随机地生成新插入结点的级别。我们注意到，在一个完全跳跃表中，具有 i 级指针的结点中有一半同时具有 $i+1$ 级指针。为了维持跳跃表的平衡性，我们可以事先确定一个实数 p，$0<p<1$，并要求在跳跃表中维持在具有 i 级指针的结点中同时具有 $i+1$ 级指针的结点所占比例约为 p。为此，在插入一个新结点时，先将其结点级别初始化为 0，然后用随机数生成器反复地产生一个[0,1) 间的随机实数 q。如果 $q<p$，则使新结点级别增加 1，直至 $q \geq p$。由此过程可知，所产生的新结点的级别为 0 的概率为 $1-p$，级别为 1 的概率为 $p(1-p)$，…，级别为 i 的概率为 $p^i(1-p)$。如此产生的新结点的级别有可能是一个很大的数，甚至远远超过表中元素的个数。为了避免这种情况，我们用 $\log_{1/p}n$ 作为新结点级别

的上界。其中，n 是当前跳跃表中结点个数。当前跳跃表中任一结点的级别不超过 $\log_{1/p}n$。在具体实现时，可用一预先确定的常数 MaxLevel 来作为跳跃表结点级别的上界。

下面我们来讨论跳跃表的实现细节。跳跃表结点类型由类 SkipNode 定义如下：

```
template < class EType, class KType > class SkipList;
template < class EType, class KType >
class SkipNode {
   friend SkipList < EType,KType > ;
   private:
     SkipNode(int size)
       {next = new SkipNode < EType,KType >  * [size];}
      ~ SkipNode() {delete [] next;}
     EType data;
     SkipNode < EType,KType >  * * next;   // 指针数组
};
```

其中，data域存放集合中元素，next是该结点的指针数组，next[i]是它的第 i 级指针。跳跃表由类 SkipList 定义如下：

```
template < class EType, class KType >
class SkipList {
   public:
     SkipList(KType Large, int MaxE = 10000, float p = 0.5);
      ~ SkipList();
     bool Search(const KType& k, EType& e) const;
     SkipList < EType,KType > & Insert(const EType& e);
     SkipList < EType,KType > & Delete(const KType& k, EType& e);
     void Output();
   private:
     int Level();
     SkipNode < EType,KType >  * SaveSearch(const KType& k);
     int MaxLevel;                              // 跳跃表级别上界
     int Levels;                                // 当前最大级别
     RandomNumber rnd;                          // 随机数产生器
     float Prob;                                // 用于分配结点级别
     KType TailKey;                             // 元素键值上界
     SkipNode < EType,KType >  * head;          // 头结点指针
     SkipNode < EType,KType >  * NIL;           // 尾结点指针
     SkipNode < EType,KType >  * * last;        // 指针数组
};
```

其中，MaxE 是集合中元素个数的上限，p 的定义如前所述。跳跃表中 0 级链元素从小到大排列。

跳跃表的构造函数初始化跳跃表的一些参数值，如 Prob，Levels，MaxLevel，TailKey 等。析构函数释放跳跃表占用的所有空间。

```
template < class EType, class KType >
```

```
SkipList<EType,KType>::SkipList(KType Large, int MaxE, float p)
{// 构造函数
   Prob = p;
   MaxLevel = ceil(log(MaxE) / log(1/p)) - 1; // 初始化跳跃表级别上界
   TailKey = Large; // 元素键值上界
   Levels = 0;  // 初始化当前最大级别
   // 创建头、尾结点和数组 last
   head = new SkipNode<EType,KType> (MaxLevel+1);
   NIL = new SkipNode<EType,KType> (0);
   last = new SkipNode<EType,KType> *[MaxLevel+1];
   NIL->data = Large;
   // 将跳跃表初始化为空表
   for (int i = 0; i <= MaxLevel; i++)
      head->next[i] = NIL;
}
```

```
template<class EType, class KType>
SkipList<EType,KType>::~SkipList()
{// 析构函数
   SkipNode<EType,KType> *next;
   // 删除所有结点
   while (head != NIL) {
      next = head->next[0];
      delete head;
      head = next;
      }
   delete NIL;
   delete [] last;
}
```

对跳跃表所表示的有序集的搜索、插入和删除等运算均要求对类 EType 进行重载，以便在 EType 与 KType 的成员间进行比较，并明确 EType 和 KType 成员间的相互赋值的定义。例如，当 EType 和 KType 分别是 int 和 long 时，其元素重载定义如下：

```
class element {
   friend void main(void);
   public:
      operator long() const {return key;}
      element& operator =(long y)
      {key = y; return *this;}
   private:
      int data;
      long key;
};
```

SkipList 类有两个搜索函数。当需要搜索集合中键值为 k 的元素时，可用共享成员函数

Search 来搜索。当 Search 搜索到键值为 k 的元素时,将该元素返回到 e 中,并返回 true,否则返回 false。算法 Search 从最高级指针链开始搜索,一直到 0 级指针链。在每一级搜索中尽可能地接近要搜索的元素。当算法从 for 循环退出时,正好处在欲寻找元素的左边。与 0 级指针所指的下一个元素进行比较,即可确定要找的元素是否在跳跃表中。

```
template< class EType, class KType>
bool SkipList< EType,KType> ::Search(const KType& k, EType& e) const
{// 搜索指定元素 k
   if (k >= TailKey) return false;
   // 位置 p 恰好位于指定元素 k 之前
   SkipNode< EType,KType>  *p = head;
   for (int i = Levels; i >= 0; i--)          // 逐级向下搜索
       while (p->next[i]->data < k)       // 在第 i 级链中搜索
          p = p->next[i];
   e = p->next[0]->data;
   return (e == k);
}
```

SkipList 的第 2 个搜索函数是私有成员函数 SaveSearch。由插入和删除操作来调用。SaveSearch 除了完成 Search 的功能外,还把每一级中遇到的上一个结点存放在数组 last 中,供插入和删除操作修改跳跃表指针时使用。

```
template< class EType, class KType>
SkipNode< EType,KType>  * SkipList< EType,KType> ::SaveSearch(const KType& k)
{// 搜索指定元素 k,并将每一级中遇到的上一个结点存放在数组 last 中
   // 位置 p 恰好位于指定元素 k 之前
   SkipNode< EType,KType>  *p = head;
   for (int i = Levels; i >= 0; i--) {
       while (p->next[i]->data < k)
          p = p->next[i];
       last[i] = p;   // 上一个第 i 级结点
       }
   return (p->next[0]);
}
```

在跳跃表中插入一个元素的算法可描述如下。在插入一个新结点时,算法随机地为其分配一个结点级别。当要插入的元素键值超过 TailKey 或表中已有相同键值的元素时,函数 Insert将引发 BadInput 异常。如果在执行插入时已没有足够的空间,则由 new 引发一个 NoMem 异常,当元素 e 被成功插入后,Insert 返回跳跃表。

```
template< class EType, class KType>
int SkipList< EType,KType> ::Level()
{// 产生不超过 MaxLevel 的随机级别
   int lev = 0;
   while (rnd.fRandom() < Prob) lev++;
   return (lev <= MaxLevel) ? lev : MaxLevel;
```

```
template < class EType, class KType >
SkipList < EType, KType > & SkipList < EType, KType > :: Insert(const EType & e)
{// 插入指定元素 e
   KType k = e; // 取得元素键值
   if (k >= TailKey) throw BadInput(); // 元素键值超界
   // 检查元素是否已存在
   SkipNode < EType, KType > *p = SaveSearch(k);
   if (p->data == e) throw BadInput(); // 元素已存在
   // 元素不存在,确定新结点级别
   int lev = Level();
   // 调整各级别指针
   if (lev > Levels) {
     for (int i = Levels + 1; i <= lev; i++)
       last[i] = head;
     Levels = lev; }
   // 产生新结点,并将新结点插入 p 之后
   SkipNode < EType, KType > *y = new SkipNode < EType, KType > (lev + 1);
   y->data = e;
   for (int i = 0; i <= lev; i++) {
      // 插入第 i 级链
      y->next[i] = last[i]->next[i];
      last[i]->next[i] = y;
      }
   return *this;
}
```

从跳跃表中删除一个元素的算法可描述如下。该算法用来删除跳跃表中键值为 k 的元素,并将所删除的元素存放在 e 中。在算法的执行过程中,若没有找到键值为 k 的元素,则引发 BadInput 异常。算法中的 while 循环用来修改 Levels 的值,找出至少包含一个元素的指针级别。当跳跃表为空时,Levels 被置为 0。

```
template < class EType, class KType >
SkipList < EType, KType > & SkipList < EType, KType > :: Delete
                                  (const KType & k, EType & e)
{// 删除键值为 k 的元素,并将所删除元素存入 e
   if (k >= TailKey) throw BadInput(); // 元素键值超界
   // 搜索待删除元素
   SkipNode < EType, KType > *p = SaveSearch(k);
   if (p->data != k) throw BadInput(); // 未找到
   // 从跳跃表中删除结点
   for (int i = 0; i <= Levels && last[i]->next[i] == p; i++)
      last[i]->next[i] = p->next[i];
   // 更新当前级别
   while (Levels > 0 && head->next[Levels] == NIL)
```

```
        Levels--;
    e = p->data;
    delete p;
    return *this;
}

template<class EType, class KType>
void SkipList<EType,KType>::Output()
{// 输出集合中元素
    SkipNode<EType,KType> *y = head->next[0];
    for (; y != NIL; y = y->next[0])
        cout << y->data << ' ';
    cout << endl;
}
```

当跳跃表中有 n 个元素时,在最坏情况下,对跳跃表进行搜索、插入和删除运算所需的计算时间均为 $O(n+\text{MaxLevel})$。在最坏情况下,可能只有一个 MaxLevel 级的元素,其余元素均在 0 级链上。此时跳跃表退化为有序链表。由于跳跃表采用了随机化技术,它的每一种运算(搜索、插入和删除)在最坏情况下的期望时间均为 $O(\log n)$。

在一般情况下,跳跃表的 1 级链上大约有 $n*p$ 个元素,2 级链上大约有 $n*p^2$ 个元素,…,i 级链上大约有 $n*p^i$ 个元素。因此跳跃表指针域占用空间的平均值是 $n\sum_i p^i = n/(1-p)$。即跳跃表所占用的空间为 $O(n)$。特别地,当 $p=0.5$ 时,约需 $2n$ 个指针空间。

7.4 拉斯维加斯(Las Vegas)算法

舍伍德型算法的优点是其计算时间复杂性对所有实例而言相对均匀。但与其相应的确定性算法相比,其平均时间复杂性没有改进。拉斯维加斯算法则不然,它能显著地改进算法的有效性。甚至对某些迄今为止找不到有效算法的问题,也能得到满意的结果。

拉斯维加斯算法的一个显著特征是它所作的随机性决策有可能导致算法找不到所需的解。因此通常用一个 bool 型函数表示拉斯维加斯型算法。当算法找到一个解时返回 true,否则返回 false。拉斯维加斯算法的典型调用形式为 bool success = LV(x, y);其中 x 是输入参数;当 success 的值为 true 时,y 返回问题的解。当 success 为 false 时,算法未能找到问题的一个解。此时可对同一实例再次独立地调用相同的算法。

设 $p(x)$ 是对输入 x 调用拉斯维加斯算法获得问题的一个解的概率。一个正确的拉斯维加斯算法应该对所有输入 x 均有 $p(x)>0$。在更强意义下,要求存在一个常数 $\delta>0$,使得对问题的每一个实例 x 均有 $p(x)\geq\delta$。设 $s(x)$ 和 $e(x)$ 分别是算法对于具体实例 x 求解成功或求解失败所需的平均时间,我们来考虑下面的算法:

```
void Obstinate(InputType x, OutputType &y)
{// 反复调用拉斯维加斯算法 LV(x,y),直到找到问题的一个解 y
    bool success = false;
    while (! success) success = LV(x,y);
```

由于 $p(x)>0$,故只要有足够的时间,对任何实例 x,上述算法 Obstinate 总能找到问题的一个解。设 $t(x)$是算法 Obstinate 找到具体实例 x 的一个解所需的平均时间,则有

$$t(x)=p(x)s(x)+(1-p(x))(e(x)+t(x))$$

解此方程可得

$$t(x)=s(x)+\frac{1-p(x)}{p(x)}e(x)$$

7.4.1 n后问题

n后问题为我们提供了设计高效的拉斯维加斯算法的很好的例子。在用回溯法解 n 后问题时,实际上是在系统地搜索整个解空间树的过程中找出满足要求的解。但我们忽略了一个重要事实:对于 n 后问题的任何一个解而言,每一个皇后在棋盘上的位置无任何规律,不具有系统性,而更像是随机放置的。由此容易想到下面的拉斯维加斯算法。我们在棋盘上相继的各行中随机地放置皇后,并注意使新放置的皇后与已放置的皇后互不攻击,直至 n 个皇后均已相容地放置好,或已没有下一个皇后的可放置位置时为止。

具体算法可描述如下。类 Queen 的私有成员 n 表示皇后个数;数组 x 存储 n 后问题的解。

```
class Queen {
  friend void nQueen(int);
  private:
    bool Place(int k);        // 测试皇后 k 置于第 x[k]列的合法性
    bool QueensLV(void);      // 随机放置 n 个皇后拉斯维加斯算法
    int n,                    // 皇后个数
        * x;                  // 解向量
};
```

类 Queen 的私有成员函数 Place(k)用于测试将皇后 k 置于第 x[k]列的合法性。

```
bool Queen::Place(int k)
{// 测试皇后 k 置于第 x[k]列的合法性
  for (int j=1;j<k;j++)
    if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k])) return false;
  return true;
}
```

类 Queen 的私有成员函数 QueensLV(void) 实现在棋盘上随机放置 n 个皇后的拉斯维加斯算法:

```
bool Queen::QueensLV(void)
{// 随机放置 n 个皇后的拉斯维加斯算法
  RandomNumber rnd;          // 随机数产生器
  int k=1;                   // 下一个放置的皇后编号
  int count=1;
```

```
    while ((k< =n) && (count>0)) {
      count=0;
      int j=0;
      for (int i=1; i< =n; i+ +) {
        x[k]=i;
        if (Place(k))
          if (rnd.Random(+ +count)= =0) j=i;   // 随机位置
        }
      if (count>0) x[k+ +]=j;
      }
    return (count>0);      // count>0 表示放置成功
  }
```

类似于算法 Obstinate，我们可以通过反复调用随机放置 n 个皇后的拉斯维加斯算法 QueensLV()，直至找到 n 后问题的一个解。

```
void nQueen(int n)
{// 解 n 后问题的拉斯维加斯算法
   Queen X;
   // 初始化 X
   X.n=n;
   int *p = new int [n+1];
   for (int i = 0; i < =n; i+ +)
       p[i] = 0;
   X.x=p;
   // 反复调用随机放置 n 个皇后的拉斯维加斯算法，直至放置成功
   while (! X.QueensLV()) ;
     for (int i=1;i< =n;i+ +)
       cout< <p[i]< < "  " ;
   cout< <endl;
   delete [] p;
}
```

上述算法一旦发现无法再放置下一个皇后，就要全部重新开始。如果将上述随机放置策略与回溯法相结合，可能会获得更好的效果。我们可以先在棋盘的若干行中随机地放置皇后，然后在后继行中用回溯法继续放置，直至找到一个解或宣告失败。随机放置的皇后越多，后继回溯搜索所需的时间就越少，但失败的概率也就越大。

与回溯法相结合的解 n 后问题的拉斯维加斯算法描述如下：

```
class Queen {
   friend void nQueen(int);
   private:
     bool Place(int k);               // 测试皇后 k 置于第 x[k]列的合法性
     void Backtrack(int t);           // 解 n 后问题的回溯法
     bool QueensLV(int stopVegas); // 随机放置 n 个皇后拉斯维加斯算法
     int n, *x, *y;
```

```
};
```

类 Queen 的私有成员函数 Place(k)用于测试将皇后 k 置于第 $x[k]$列的合法性。
类 Queen 的私有成员函数 Backtrack(t)是解 n 后问题的回溯法。

```
bool Queen::Place(int k)
{// 测试皇后 k 置于第 x[k]列的合法性
  for (int j = 1;j < k;j ++)
    if ((abs(k - j) == abs(x[j] - x[k]))||(x[j] == x[k])) return false;
  return true;
}

void Queen::Backtrack(int t)
{// 解 n 后问题的回溯法
  if (t > n) {
    for (int i= 1;i <= n;i ++)
      y[i] = x[i];
    return;
    }
    else
      for (int i = 1;i <= n;i ++) {
        x[t] = i;
        if (Place(t)) Backtrack(t + 1);

 }
```

类 Queen 的私有成员函数 QueensLV(stopVegas) 实现在棋盘上随机放置若干个皇后的拉斯维加斯算法。其中，$1 \leq stopVegas \leq n$ 表示随机放置的皇后数。

```
bool Queen::QueensLV(int stopVegas)
{ // 随机放置 n 个皇后拉斯维加斯算法
  RandomNumber rnd;
  int k = 1;  // 随机数产生器
  int count = 1;
  // 1≤stopVegas≤n 表示允许随机放置的皇后数
  while ((k <= stopVegas) && (count > 0)) {
    count = 0;
    int j = 0;
    for (int i = 1; i <= n; i ++) {
      x[k] = i;
      if (Place(k))
        if (rnd.Random(++count) == 0) j = i; // 随机位置
      }
    if (count > 0) x[k ++] = j;
    }
  return (count > 0);        // count > 0 表示放置成功
```

算法的回溯搜索部分与解 n 后问题的回溯法类似,所不同的是这里只要找到一个解就可以了。

```
void nQueen(int n)
{// 与回溯法相结合的解 n 后问题的拉斯维加斯算法
    Queen X;
    // 初始化 X
    X.n = n;
    int *p = new int [n+1];
    int *q = new int [n+1];
    for (int i = 0; i <= n; i++) {
        p[i] = 0;
        q[i] = 0;}
    X.y = p;
    X.x = q;
    int stop = 5;
    while (! X.QueensLV(stop)) ;
    // 算法的回溯搜索部分
    X.Backtrack(stop + 1);
    for (int i = 1;i <= n;i++)
      cout << p[i] << " ";
    cout << endl;
    delete [] p;
    delete [] q;
}
```

下面的表7-1 给出了用上述算法解 8 后问题时,对于不同的 stopVegas 取值,算法成功的概率 p,一次成功搜索访问的结点数平均值 s,一次不成功搜索访问的结点数平均值 e,以及反复调用算法使得最终找到一个解所访问的结点数的平均值 $t = s + (1 - p)e/p$。

表7-1 解 8 后问题的拉斯维加斯算法中不同 stopVegas 值所相应的算法效率

stopVegas	p	s	e	t
0	1.0000	114.00	—	114.00
1	1.0000	39.63	—	39.63
2	0.8750	22.53	39.67	28.20
3	0.4931	13.48	15.10	29.01
4	0.2618	10.31	8.79	35.10
5	0.1624	9.33	7.29	46.92
6	0.1375	9.05	6.98	53.50
7	0.1293	9.00	6.97	55.93
8	0.1293	9.00	6.97	55.93

stopVegas = 0 相应于完全使用回溯法的情形。

表7-2 是当 $n=12$ 时,关于若干 stopVegas 值的统计数据。由此可以看出当 $n=12$ 时,取 stopVegas = 5 时,算法效率很高。

表7-2 解 12 后问题的拉斯维加斯算法中不同 stopVegas 值所相应的算法效率

stopVegas	p	s	e	t
0	1.0000	262.00	-	262.00
5	0.5039	33.88	47.23	80.39
12	0.0465	13.00	10.20	222.11

7.4.2 整数因子分解

设 $n>1$ 是一个整数。关于整数 n 的因子分解问题是,找出 n 的如下形式的惟一分解式:

$$n = p_1^{m_1} p_2^{m_2} \cdots p_k^{m_k}$$

其中,$p_1 < p_2 < \cdots < p_k$ 是 k 个素数,$m_1, m_2, \cdots, m_k$ 是 k 个正整数。

如果 n 是一个合数,则 n 必有一个非平凡因子 x,$1<x<n$,使得 x 可以整除 n。给定一个合数 n,求 n 的一个非平凡因子的问题称为整数 n 的因子分割问题。

在本章的下一节中我们会讨论一个用于测试给定整数的素性的蒙特卡罗算法。有了测试素性的算法后,整数的因子分解问题就转化为整数的因子分割问题。

下面的算法 Split(n)可实现对整数的因子分割。

```
int split(int n)
{
  int m = floor(sqrt(double(n)));
  for (int i=2; i<=m; i++)
    if (n%i==0) return i;
  return 1;
}
```

在最坏情况下,算法 Split(n)所需的计算时间为 $\Omega(\sqrt{n})$。当 n 较大时,上述算法无法在可接受的时间内完成因子分割任务。对于给定的正整数 n,设其位数为 $m=\lceil \log_{10}(1+n) \rceil$。由$\sqrt{n}=\theta(10^{m/2})$知,算法 Split($n$)是关于 m 的指数时间算法。

到目前为止,还没有找到解因子分割问题的多项式时间算法。事实上,算法 Split(n)是对范围在 $1\sim x$ 的所有整数进行了试除而得到范围在 $1\sim x^2$ 的任一整数的因子分割。下面我们要讨论的求整数 n 的因子分割的拉斯维加斯算法是由 Pollard 提出的,该算法的效率比算法 Split(n)有较大的提高。Pollard 算法用与算法 Split(n)相同的工作量就可以得到在 $1\sim x^4$ 范围内整数的因子分割。

Pollard 算法在开始时选取 $0\sim(n-1)$范围内的随机数 x_1,然后递归地由

$$x_i = (x_{i-1}^2 - 1) \bmod n$$

产生无穷序列 $x_1, x_2, \cdots, x_k, \cdots$。

对于 $i=2^k$、$k=0,1,\cdots$,以及 $2^k < j \leq 2^{k+1}$,算法计算出 $x_j - x_i$ 与 n 的最大公因子

$$d = \gcd(x_j - x_i, n)$$

如果 d 是 n 的非平凡因子,则实现对 n 的一次分割,算法输出 n 的因子 d。

求整数 n 因子分割的拉斯维加斯算法 Pollard(n)可描述如下。其中,gcd(a, b)是求 2 个

整数最大公因数的欧几里得算法。

```
int gcd(int a, int b)
{// 求整数 a 和 b 最大公因数的欧几里得算法
  if (b = = 0) return a;
  else return gcd(b,a%b);
}
```

```
void Pollard(int n)
{// 求整数 n 因子分割的拉斯维加斯算法
  RandomNumber rnd;
  int i = 1;
  int x = rnd.Random(n); // 随机整数
  int y = x;
  int k = 2;
  while (true) {
    i+ +;
    x = (x * x - 1)%n;      //  xi = (x²i-1 - 1)modn
    int d = gcd(y - x,n);   // 求 n 的非平凡因子
    if ((d > 1) & & (d < n)) cout< < d< < endl;
    if (i = = k) {
      y = x;
      k * = 2;}
    }
}
```

对 Pollard 算法更深入的分析可知,执行算法的 while 循环约$\sqrt{p}$次后,Pollard 算法会输出 n 的一个因子 p。由于 n 的最小素因子 $p \leqslant \sqrt{n}$,故 Pollard 算法可在 $O(n^{1/4})$时间内找到 n 的一个素因子。

在上述 Pollard 算法中还可将产生序列 x_i 的递归式改作

$$x_i = (x_{i-1}^2 - c) \bmod n$$

其中,c 是一个不等于 0 和 2 的整数。

7.5 蒙特卡罗(Monte Carlo)算法

在实际应用中我们常会遇到一些问题,不论采用确定性算法或概率算法都无法保证每次都能得到正确的解答。蒙特卡罗算法则在一般情况下可以保证对问题的所有实例都以高概率给出正确解,但是通常无法判定一个具体解是否正确。

7.5.1 蒙特卡罗算法的基本思想

设 p 是一个实数,且 $1/2 < p < 1$。如果一个蒙特卡罗算法对于问题的任一实例得到正确解的概率不小于 p,则称该蒙特卡罗算法是 p 正确的,且称 $p - 1/2$ 是该算法的优势。

如果对于同一实例,蒙特卡罗算法不会给出两个不同的正确解答,则称该蒙特卡罗算法是

一致的。

有些蒙特卡罗算法除了具有描述问题实例的输入参数外,还具有描述错误解可接受概率的参数。这类算法的计算时间复杂性通常由问题的实例规模以及错误解可接受概率的函数来描述。

对于一个一致的 p 正确蒙特卡罗算法,要提高获得正确解的概率,只要执行该算法若干次,并选择出现频次最高的解即可。

在一般情况下,设 ε 和 δ 是两个正实数,且 $\varepsilon+\delta<1/2$。设 MC(x)是一个一致的($1/2+\varepsilon$)正确的蒙特卡罗算法,且 $C_\varepsilon=-2/\log(1-4\varepsilon^2)$。如果我们调用算法 MC($x$)至少 $\lceil C_\varepsilon\log 1/\delta\rceil$ 次,并返回各次调用出现频数最高的解,就可以得到解同一问题的一个一致的($1-\delta$)正确的蒙特卡罗算法。由此可见,不论算法 MC(x)的优势 $\varepsilon>0$ 多小,我们都可以通过反复调用来放大算法的优势,使得最终得到的算法具有可接受的错误概率。

要证明上述论断,设 $n>C_\varepsilon\log 1/\delta$ 是重复调用($1/2+\varepsilon$)正确的算法 MC(x)的次数,且 $p=(1/2+\varepsilon)$, $q=1-p=(1/2-\varepsilon)$, $m=\lfloor n/2\rfloor+1$。经 n 次反复调用算法 MC(x),找到问题的一个正确解。则该正确解至少应出现 m 次,因此其出现错误概率最多是

$$
\begin{aligned}
&\sum_{i=0}^{m-1}\mathrm{Prob}\{n\text{ 次调用出现 }i\text{ 次正确解}\}\\
&\leqslant \sum_{i=0}^{m-1}\binom{n}{i}p^iq^{n-i}\\
&=(pq)^{n/2}\sum_{i=0}^{m-1}\binom{n}{i}(q/p)^{n/2-i}\\
&\leqslant (pq)^{n/2}\sum_{i=0}^{m-1}\binom{n}{i} \qquad (\text{由于 } q/p<1,\text{且 } n/2-i\geqslant 0)\\
&\leqslant (pq)^{n/2}\sum_{i=0}^{n}\binom{n}{i}\\
&=(pq)^{n/2}2^n\\
&=(4pq)^{n/2}\\
&=(1-4\varepsilon^2)^{n/2}\\
&\leqslant (1-4\varepsilon^2)^{(c_\varepsilon/2)\log(1/\delta)} \qquad (\text{由于 } 0<(1-4\varepsilon^2)<1)\\
&=2^{-\log(1/\delta)}\\
&=\delta \qquad (\text{由于对任意 } x>0 \text{ 有 } x^{1/\log x}=2)
\end{aligned}
$$

由此可知重复 n 次调用算法 MC(x) 得到正确解的概率至少为 $1-\delta$。

更进一步的分析表明,如果重复调用一个一致的($1/2+\varepsilon$) 正确的蒙特卡罗算法 $2m-1$ 次,得到正确解的概率至少为 $1-\delta$,其中,

$$\delta=\frac{1}{2}-\varepsilon\sum_{i=0}^{m-1}\binom{2i}{i}\left(\frac{1}{4}-\varepsilon^2\right)^i\leqslant\frac{(1-4\varepsilon^2)^m}{4\varepsilon\sqrt{\pi m}}$$

在实际使用中,大多数蒙特卡罗算法经重复调用后正确率提高很快。

设 MC(x) 是解某个判定问题 D 的蒙特卡罗算法。当 MC(x) 返回 true 时解总是正确的,仅当它返回 false 时有可能产生错误的解。我们称这类的蒙特卡罗算法为偏真算法。

显而易见,当多次调用一个偏真蒙特卡罗算法时,只要有一次调用返回 true,就可以断定

相应的解为 true。稍后我们将看到,在这种情况下,只要重复调用偏真蒙特卡罗算法 4 次,就可以将解的正确率从55% 提高到95% ,重复调用算法6次,可将解的正确率提高到99%。而且对于偏真蒙特卡罗算法而言,原来对 p 正确算法的要求 $p > 1/2$ 可以放松为 $p > 0$ 即可。

现在我们回到一般问题,即所讨论的问题不一定是一个判定问题。设 y_0 是所求解问题的一个特殊的解答,如判定问题的 true 解答。对于一个解所给问题的蒙特卡罗算法 MC(x),如果存在问题实例的子集 X 使得:

(1) 当 $x \notin X$ 时,MC(x) 返回的解是正确的;

(2) 当 $x \in X$ 时,正确解是 y_0,但 MC(x) 返回的解未必是 y_0。

我们称上述算法 MC(x) 是偏 y_0 的算法。

设 MC(x) 是一个一致的,p 正确偏 y_0 蒙特卡罗算法. MC(x) 返回的解为 y_0。我们来讨论以下两种情形:

(1) $y = y_0$ 的情形:

此时,MC(x) 返回的解是正确的.

事实上,当 $x \notin X$ 时,MC(x) 返回的解总是正确的。当 $x \in X$ 时,正确解是 y_0,故此时,算法返回的解也是正确的。

(2) $y \neq y_0$ 的情形:

在这种情形下,当 $x \notin X$ 时,y 是正确的。当 $x \in X$ 时,y 是错误的。因为此时正确解是 y_0,而 $y \neq y_0$。但是由于算法是 p 正确的,产生这种错误的概率不超过 $1 - p$。

在一般情况下,如果重复 k 次调用 MC(x),所返回的解依次为 $y_1, \cdots, y_k$,则

① 存在 i 使 $y_i = y_0$,此时 y_0 为正确解;

② 存在 $i \neq j$,使得 $y_i \neq y_j$,此时必有 $x \in X$,因此可知正确解为 y_0;

③ 对所有 i 有 $y_i = y$,但 $y \neq y_0$,此时,正确解仍有可能是 y_0。

如果情形 3) 发生,则每一次调用 MC(x) 均产生错误解 y,但发生这种情况的概率不超过 $(1 - p)^k$。

由上面的讨论可知,重复调用一个一致的,p 正确偏 y_0 蒙特卡罗算法 k 次,可得到一个 $(1 - (1 - p)^k)$ 正确的蒙特卡罗算法,且所得算法仍是一个一致的偏 y_0 蒙特卡罗算法。特别地,调用一个偏真蒙特卡罗算法 k 次可将其正确概率从 p 提高到 $(1 - (1 - p)^k)$。

7.5.2 主元素问题

设 T[1:n] 是一个含有 n 个元素的数组。当 $|\{i \mid T[i] = x\}| > n/2$ 时,称元素 x 是数组 T 的主元素。对于给定的输入数组 T,考虑下面判定所给数组 T 是否含有主元素的蒙特卡罗算法 Majority。

```
RandomNumber rnd;
template < class Type >
bool Majority(Type * T, int n)
{// 判定主元素的蒙特卡罗算法
    int i = rnd.Random(n) + 1;
    Type x = T[i];      // 随机选择数组元素
    int k = 0;
```

```
    for (int j = 1;j <= n;j++)
      if (T[j] == x) k++;
    return (k > n/2);   // k > n/2 时 T 含有主元素
}
```

上述算法对随机选择的数组元素 x，测试它是否为数组 T 的主元素。如果算法返回的结果为 true，则随机选择的数组元素 x 是数组 T 的主元素，显然数组 T 含有主元素。反之，如果算法返回的结果为 false，则数组 T 未必没有主元素。可能数组 T 含有主元素，而随机选择的数组元素 x 不是 T 的主元素。由于数组 T 的非主元素个数小于 $n/2$，故上述情况发生的概率小于 1/2。由此可见上述判定数组 T 的主元素存在性算法是一个偏真的 1/2 正确算法，或换句话说，如果数组 T 含有主元素，则算法以大于 1/2 的概率返回 true；如果数组 T 没有主元素，则算法肯定返回 false。

在实际使用时，50% 的错误概率是不可容忍的。使用前面讨论过的重复调用技术可将错误概率降低到任何可接受值的范围内。首先我们来看重复调用 2 次的算法 Majority2 如下：

```
template<class Type>
bool Majority2(Type *T, int n)
{// 重复 2 次调用算法 Majority
    if (Majority(T,n)) return true;
    else return Majority(T,n);
}
```

如果数组 T 不含主元素，则每次调用 Majority(T, n)返回的值肯定是 false，从而 Majority2 返回的值肯定也是 false。如果数组 T 含有主元素，则算法 Majority(T, n)返回 true 的概率 p 大于 1/2，而当 Majority(T, n)返回 true 时，Majority2 也返回 true。另一方面，Majority2 的第一次调用 Majority(T, n)返回 false 的概率为 $1-p$，第二次调用 Majority(T, n)仍以概率 p 返回 true。因此当数组 T 含有主元素时，Majority2 返回 true 的概率是 $p+(1-p)p=1-(1-p)^2>3/4$。也就是说，算法 Majority2 是一个偏真 3/4 正确的蒙特卡罗算法。

算法 Majority2 中，重复调用 Majority(T, n)所得到的结果是相互独立的。当数组 T 含有主元素时，某次调用 Majority(T, n)返回 false 并不会影响下一次调用 Majority(T, n)返回值为 true 的概率。因此，k 次重复调用 Majority(T, n)均返回 false 的概率小于 2^{-k}。另一方面，在 k 次调用中，只要有一次调用返回的值为 true，即可断定数组 T 含有主元素。

对于任何给定的 $\varepsilon>0$，下面的算法 MajorityMC 重复调用$\lceil\log(1/\varepsilon)\rceil$次算法 Majority。它是一个偏真蒙特卡罗算法，且其错误概率小于 ε。

```
template<class Type>
bool MajorityMC(Type *T, int n, double e)
{// 重复⌈log(1/ε)⌉次调用算法 Majority
    int k = ceil(log(1/e)/log(2));
    for (int i = 1;i <= k;i++)
      if (Majority(T,n)) return true;
    return false;
}
```

算法 MajorityMC 所需的计算时间显然是 $O(n\log(1/\varepsilon))$。

7.5.3 素数测试

关于素数的研究已有相当长的历史，近代密码学的研究又给它注入了新的活力。在关于素数的研究中素数的测试是一个非常重要的问题。Wilson 定理给出了一个数是素数的充要条件。

Wilson 定理 对于给定的正整数 n，判定 n 是一个素数的充要条件是

$$(n-1)! \equiv -1(\bmod n)$$

Wilson 定理有很高的理论价值。但实际用于素性测试所需计算量太大，无法实现对较大素数的测试。到目前为止，尚未找到素数测试的有效的确定性算法或拉斯维加斯型算法。

首先容易想到下面的素数测试概率算法 Prime。

```
bool Prime(unsigned int n)
{
    RandomNumber rnd;
    int m = floor(sqrt(double(n)));
    unsigned int a = rnd.Random(m - 2) + 2;
    return (n%a! =0);
}
```

算法 Prime 返回 false 时，算法幸运地找到 n 的一个非平凡因子，因此可以肯定 n 是一个合数。但是对于上述算法 Prime 来说，即使 n 是一个合数，算法仍以高概率返回 true。例如，当 $n = 2\,623 = 43 \times 61$ 时，算法 Prime 在 2 ~ 51 范围内随机选择一个整数 a，仅当选择到 $a = 43$ 时，算法返回 false，其余情况均返回 true。在 2 ~ 51 范围内选到 $a = 43$ 的概率约为 2%，因此算法以 98% 的概率返回错误的结果 true。当 n 增大时，情况就更糟。当然在上述算法中可以用欧几里得算法判定 n 与 a 是否互素来提高测试效率，但结果仍不理想。

著名的费尔马小定理为素数判定提供了一个有力的工具。

费尔马小定理 如果 p 是一个素数，且 $0 < a < p$，则 $a^{p-1} \equiv 1(\bmod p)$。

例如，67 是一个素数，则 $2^{66}\bmod 67 = 1$。

利用费尔马小定理，对于给定的整数 n，可以设计一个素数判定算法。通过计算 $d = 2^{n-1}\bmod n$ 来判定整数 n 的素性。当 $d \neq 1$ 时，n 肯定不是素数；当 $d = 1$ 时，n 则很可能是素数。但也存在合数 n 使得 $2^{n-1} \equiv 1(\bmod n)$。例如，满足此条件的最小合数是 $n = 341$。为了提高测试的准确性，我们可以随机地选取整数 $1 < a < n-1$，然后用条件 $a^{n-1} \equiv 1(\bmod n)$ 来判定整数 n 的素性。例如对于 $n = 341$，取 $a = 3$ 时，有 $3^{340} \equiv 56(\bmod 341)$。故可判定 n 不是素数。

费尔马小定理毕竟只是素数判定的一个必要条件。满足费尔马小定理条件的整数 n 未必全是素数。有些合数也满足费尔马小定理的条件。这些合数被称作 Carmichael 数，前 3 个 Carmichael 数是 561，1 105 和 1 729。Carmichael 数是非常少的。在 1 ~ 100 000 000 范围内的整数中，只有 255 个 Carmichael 数。

利用下面的二次探测定理可以对上面的素数判定算法作进一步改进，以避免将 Carmichael 数当作素数。

二次探测定理 如果 p 是一个素数，且 $0 < x < p$，则方程 $x^2 \equiv 1(\bmod p)$ 的解为

$x = 1, p - 1$。

事实上，$x^2 \equiv 1(\mathrm{mod}\ p)$ 等价于 $x^2 - 1 \equiv 0(\mathrm{mod}\ p)$。由此可知，

$$(x - 1)(x + 1) \equiv 0(\mathrm{mod}\ p)$$

故 p 必须整除 $x - 1$ 或 $x + 1$。由 p 是素数且 $0 < x < p$ 推出 $x = 1$ 或 $x = p - 1$。

利用二次探测定理，我们可以在利用费尔马小定理计算 $a^{n-1} \mathrm{mod}\ n$ 的过程中增加对于整数 n 的二次探测。一旦发现违背二次探测条件，即可得出 n 不是素数的结论。

下面的算法 power 用于计算 $a^p \mathrm{mod}\ n$，并在计算过程中实施对 n 的二次探测。

```
void power( unsigned int a, unsigned int p, unsigned int n,
            unsigned int &result, bool &composite)
{// 计算 a^p mod n,并实施对 n 的二次探测
  unsigned int x;
  if (p = = 0) result = 1;
  else {
    power(a,p/2,n,x,composite);   // 递归计算
    result = (x * x)%n;            // 二次探测
    if ((result = = 1)&&(x! = 1)&&(x! = n-1))
      composite = true;
    if ((p%2) = = 1)         // p 是奇数
      result = (result * a)%n;
    }
}
```

在算法 power 的基础上，可设计素数测试的蒙特卡罗算法 Prime 如下：

```
bool Prime(unsigned int n)
{// 素数测试的蒙特卡罗算法
   RandomNumber rnd;
   unsigned int a, result;
   bool composite = false;
   a = rnd.Random(n-3)+2;
   power(a,n-1,n,result,composite);
   if (composite||(result! = 1)) return false;
   else return true;
```

算法 Prime 返回 false 时，整数 n 一定是一个合数。而当算法 Prime 返回值为 true 时，整数 n 在高概率意义下是一个素数。仍然可能存在合数 n，对于随机选取的基数 a，算法返回 true。但对于上述算法的深入分析表明，当 n 充分大时，这样的基数 a 不超过 $(n-9)/4$ 个，由此可知上述算法 Prime 是一个偏假 3/4 正确的蒙特卡罗算法。

正如我们前面讨论过的，上述算法 Prime 的错误概率可通过多次重复调用而迅速降低。重复 k 次调用算法 Prime 的蒙特卡罗算法 PrimeMC 可描述如下：

```
bool PrimeMC(unsigned int n,unsigned int k)
{// 重复 k 次调用算法 Prime 的蒙特卡罗算法
  RandomNumber rnd;
```

```
        unsigned int a, result;
        bool composite = false;
        for (int i = 1;i <= k;i++) {
          a = rnd.Random(n - 3) + 2;
          power(a,n - 1,n,result,composite);
          if (composite||(result != 1)) return false;
          }
        return true;
      }
```

易知算法 PrimeMC 的错误概率不超过$(1/4)^k$。这是一个很保守的估计,实际使用的效果要好得多。

习题 7

7-1　在实际应用中,常需模拟服从正态分布的随机变量,其密度函数为

$$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-a)^2}{2\sigma^2}}$$

其中,a 为均值,σ 为标准差.

如果 s 和 t 是$(-1,1)$中均匀分布的随机变量,且 $s^2+t^2<1$,令

$$p = s^2 + t^2$$
$$q = \sqrt{(-2\ln p)/p}$$
$$u = sq$$
$$v = tq$$

则 u 和 v 是服从标准正态分布($a=0,\sigma=1$)的 2 个互相独立的随机变量。

(1) 利用上述事实,设计一个模拟标准正态分布随机变量的算法。

(2) 将上述算法扩展到一般的正态分布。

7-2　设有一个文件含有 n 个记录。

(1) 试设计一个算法随机抽取该文件中 m 个记录。

(2) 如果事先不知道文件中记录个数,应如何随机抽取其中的 m 个记录。

7-3　试设计一个算法随机地产生范围在 $1\sim n$ 中的 m 个随机整数,且要求这 m 个随机整数互不相同。

7-4　设 X 是一个含有 n 个元素的集合,从 X 中均匀地选取元素。设第 k 次选取时首次出现重复。

(1) 试证明当 n 充分大时,k 的期望值为$\beta\sqrt{n}$,其中,$\beta=\sqrt{\pi/2}=1.253$。

(2) 由此设计一个计算给定集合 X 中元素个数的概率算法。

7-5　试设计一个概率算法计算 $365!/340!365^{25}$,并精确到 4 位有效数字。

7-6　一个问题是易验证的是指对该问题的给定实例的每一个解,都可以有效地验证其正确性。例如求一个整数的非平凡因子问题是易验证的,而求一个整数的最小非平凡因子就不是易验证的。在一般情况下,易验证问题未必是易解的。

(1) 给定一个解易验证问题P的蒙特卡罗方法,由此设计一个相应的解问题P的拉斯维加

斯算法

（2）给定一个解易验证问题P的拉斯维加斯算法，由此设计一个相应的解问题P的蒙特卡罗算法。

7-7　用数组模拟有序链表的数据结构，设计支持下列运算的舍伍德型算法，并分析各种运算所需的计算时间：

（1）Predecessor　找出一给定元素 x 在有序集 S 中的前驱元素；

（2）Successor　找出一给定元素 x 在有序集 S 中的后继元素；

（3）Min　找出有序集 S 中的最小元素；

（4）Max　找出有序集 S 中的最大元素。

7-8　采用数组模拟有序链表的数据结构，设计一个舍伍德型排序算法，使算法最坏情况下的平均计算时间为 $O(n^{3/2})$。

7-9　如果对于某一个 n 的值，n后问题无解，则算法将陷入死循环。

（1）证明或否定下述论断：对于 $n \geqslant 4$，n后问题有解。

（2）是否存在一个正数 δ，使得对所有 $n \geqslant 4$ 算法成功的概率至少是 δ。

7-10　设 p 是一个奇素数，$1 \leqslant x \leqslant p-1$，如果存在一个整数 y，$1 \leqslant y \leqslant p-1$，使得 $x \equiv y^2 (\bmod\ p)$，则称 y 是 x 的模 p 平方根。例如63是55的模103平方根。试设计一个求整数 x 的模 p 平方根的拉斯维加斯算法。

7-11　假设已有一个算法 Prime(n) 可用于测试整数 n 是否为一素数。另外还有一个算法 Split(n) 可以实现对合数 n 的因子分割。试利用这两个算法设计一个对给定整数 n 进行因子分解的算法。

7-12　（1）试证明下面的算法Primality能以80%以上的正确率判定给定的一个整数 n 是否为素数。另一方面，举出整数 n 的一个例子表明算法对此整数 n 总是给出错误的解答，进而说明该算法不是一个蒙特卡罗算法。

```
bool Primality(int n)
{
  if (gcd(n,30030) == 1) return true;
  else return false;
}
```

（2）试找出上述算法 Primality 中可用于替换整数30 030的另一个整数，使得用此整数代替30 030后，算法的正确率提高到85%以上，且允许整数 n 是非常大的整数。

7-13　设 MC(x) 是一个一致的75%正确的蒙特卡罗算法，考虑下面的算法

```
MC3(x)
{
  t = MC(x);
  u = MC(x);
  v = MC(x);
  if ((t == u) || (t == v)) return t;
  return v;
```

}

(1) 试证明上述算法 MC3(x) 是一致的 27/32 正确的算法,因此是 84% 正确的。

(2) 试证明如果 MC(x) 不是一致的,则 MC3(x) 的正确率有可能低于 71%。

7-14　设 $I=\{1,2,\cdots,n\}$, $S\subseteq I$ 是 I 的一个子集。MC(x) 是一个偏假 p 正确蒙特卡罗算法。该算法用于判定所给的整数 $1\leqslant x\leqslant n$ 是否为集合 S 中的整数,即 $x\in S$。设 $q=1-p$。由偏假算法的定义可知,对任意 $x\in S$ 有 $\mathrm{Prob}\{\mathrm{MC}(x)=\mathrm{true}\}=1$。当 $x\notin S$ 时, $\mathrm{Prob}\{\mathrm{MC}(x)=\mathrm{true}\}\leqslant q$。考虑下面的产生 S 中随机元素的算法 GenRand 如下:

```
bool RepeatMC(int x,int k)
{
  int i = 0;
  bool ans = true;
  while (ans&&(i < k)) {
    i++;
    ans = MC(x);
    }
  return ans;
}
```

```
int GenRand(int n,int k)
{
  RandomNumber rnd;
  int x = rnd.Random(n) + 1;
  while (!RepeatMC(x,k)) x = rnd.Random(n) + 1;
  return x;
}
```

假设由语句 x = rnd.Random(n) + 1 产生的整数 $x\in S$ 的概率为 r,证明算法 GenRand 返回的整数不在 S 中的概率最多为

$$\frac{1}{1+\frac{r}{1-r}q^{-k}}$$

7-15　设算法 A 和 B 是解同一判定问题的两个有效的蒙特卡罗算法。算法 A 是一个 p 正确偏真算法,算法 B 则是一个 q 正确偏假算法。试利用这两个算法设计一个解同一问题的拉斯维加斯算法,并使所得到的算法对任何实例的成功率尽可能高。

7-16　考虑下面的无限循环算法:

```
void PrintPrimes(void)
{
   cout << '2' << endl;
   cout << '3' << endl;
   int n = 5;
   while (true) {
     int m = floor(log(double(n)));
```

```
        if (PrimeMC(n,m)) cout << n << endl;
        n = n + 2;
        }
    }
```

易知,每一个素数都会被上述算法输出。但是除了所有素数外,算法可能偶尔错误地输出某些合数。说明上述情况不太少可能发生。或更精确地,证明上述算法错误地输出一个合数的概率小于 1%。

7-17　试设计一个素数测试的偏真蒙特卡罗算法。要求对于测试的整数 n,所述算法是一个关于 $\log n$ 的多项式时间算法。

结合教材中素数测试的偏假蒙特卡罗算法,设计一个素数测试的拉斯维加斯算法(参见习题7-15)。

7-18　给定两个集合 S 和 T,试设计一个判定 S 和 T 是否相等的蒙特卡罗算法。

7-19　给定三个 $n \times n$ 矩阵 $\boldsymbol{A}$,$\boldsymbol{B}$ 和 $\boldsymbol{C}$,下面的偏假 1/2 正确的蒙特卡罗算法用于判定 $\boldsymbol{AB} = \boldsymbol{C}$。

```
bool Product(int **A, int **B, int **C, int n)
{// 判定 AB = C 的蒙特卡罗算法
  RandomNumber rnd;
  int *x = new int [n + 1];
  int *y = new int [n + 1];
  int *z = new int [n + 1];
  for (int i = 1;i <= n;i++) {
    x[i] = rnd.Random(2);
    if (x[i] == 0) x[i] = -1;
    }
  Mult(B,x,y,n);
  Mult(A,y,z,n);
  Mult(C,x,y,n);
  for (int i = 1;i <= n;i++)
    if (y[i]!= z[i]) return false;
  return true;
}
```

算法所需的计算时间为 $O(n^2)$。显然,当 $\boldsymbol{AB} = \boldsymbol{C}$ 时,算法 Product($\boldsymbol{A}$,$\boldsymbol{B}$,$\boldsymbol{C}$,n) 返回 true。试证明当 $\boldsymbol{AB} \neq \boldsymbol{C}$ 时,算法返回值为 false 的概率至少为 1/2(提示:考虑矩阵 $\boldsymbol{AB} - \boldsymbol{C}$ 并证明当 $\boldsymbol{AB} \neq \boldsymbol{C}$ 时,将该矩阵各行相加或相减最终得到的行向量至少有一半是非零向量)。

7-20　给定两个 $n \times n$ 矩阵 $\boldsymbol{A}$,$\boldsymbol{B}$,试设计一个判定 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 是否互逆的蒙特卡罗算法。

7-21　给定阶数分别为 n,n 和 $2n$ 的多项式 $p(x)$,$q(x)$ 和 $r(x)$。试设计一个判定 $p(x)q(x) = r(x)$ 的偏假 1/2 正确的蒙特卡罗算法,并要求算法的计算时间为 $O(n)$。

第 8 章　线性规划与网络流

学习要点

· 理解线性规划算法模型
· 掌握解线性规划问题的单纯形算法
· 理解网络与网络流的基本概念
· 掌握网络最大流的增广路算法
· 掌握网络最大流的预流推进算法
· 掌握网络最小费用流的消圈算法
· 掌握网络最小费用流的最小费用路算法
· 掌握网络最小费用流的网络单纯形算法

8.1　线性规划问题和单纯形算法

8.1.1　线性规划问题及其表示

线性规划问题可表示为如下形式：

$$\max \sum_{j=1}^{n} c_j x_j \tag{8.1}$$

$$\text{s.t.} \quad \sum_{t=1}^{n} a_{it} x_t \leqslant b_i \quad (i = 1,2,\cdots,m_1) \tag{8.2}$$

$$\sum_{t=1}^{n} a_{jt} x_t = b_j \quad (j = m_1 + 1,\cdots,m_1 + m_2) \tag{8.3}$$

$$\sum_{t=1}^{n} a_{kt} x_t \geqslant b_k \quad (k = m_1 + m_2 + 1,\cdots,m_1 + m_2 + m_3) \tag{8.4}$$

$$x_t \geqslant 0 \quad (t = 1,2,\cdots,n) \tag{8.5}$$

上面各式中，$x_1,x_2,\cdots,x_n$ 是 n 个独立变量。(8.1) 式是线性规划问题的目标函数。max 是 maximize 的缩写，表示求极大值。稍后将看到求目标函数极小值 min 的线性规划问题很容易转换为与之等价的求目标函数极小值的线性规划问题。(8.2) ~ (8.5) 式是线性规划问题的约束条件。s.t.是 subject to 的缩写，表示“满足于”。(8.2) 式有 m_1 个不等式($\leqslant$) 约束；(8.3) 式有 m_2 个等式约束；(8.4) 式有 m_3 个不等式($\geqslant$) 约束。(8.2) ~ (8.4) 式约束总个数为 $m = m_1 + m_2 + m_3$ (8.2) ~ (8.4) 式中系数 a_{ij} 可正可负，也可以是零。而所有约束的右端参数规定为非负数，即 $b_j \geqslant 0, j = 1,2,\cdots,m$，但这只是一种约定而已，因为可以用 -1 去乘任何一个约束的两端。(8.5) 式是线性规划问题的变量非负性约束条件。

变量 $x_1,x_2,\cdots,x_n$ 满足约束条件(8.2) ~ (8.5) 式的一组值称为线性规划问题的一个可行解。所有可行解构成的集合称为线性规划问题的可行区域。使目标函数取得极值的可行解称为最优解。在最优解处目标函数的值称为最优值。

有些情况下可能不存在最优解。通常有两种情况：

(1) 根本没有可行解，即给定的约束条件之间是相互排斥的，可行区域为空集。

(2) 目标函数没有极值。也就是说，在 n 维空间的某个方向上，目标函数值可以无限增大，而仍满足约束条件，此时目标函数值无界。

下面给出线性规划问题的一个具体例子。

$$\max z = x_1 + x_2 + 3x_3 - x_4 \tag{8.6}$$

$$\text{s.t.} \quad \begin{aligned} & x_1 + 2x_3 \leqslant 18 \\ & 2x_2 - 7x_4 \leqslant 0 \\ & x_1 + x_2 + x_3 + x_4 = 9 \\ & x_2 - x_3 + 2x_4 \geqslant 1 \\ & x_i \geqslant 0 \ (i = 1,2,3,4) \end{aligned} \tag{8.7}$$

此例中，$n = 4, m_1 = 2, m_2 = m_3 = 1, m = m_1 + m_2 + m_3 = 4$。

这个问题的解为 $(x_1, x_2, x_3, x_4) = (0,3.5,4.5,1)$；最优值为 16。下面将详细讨论如何求解。

8.1.2 线性规划基本定理

使约束条件(8.2) ~ (8.5)式中的某 n 个约束以等号满足的可行解称为线性规划问题的基本可行解。若 $n > m$，则基本可行解中至少有 $n - m$ 个分量为0。也就是说，基本可行解中最多有 m 个分量非零。

线性规划基本定理 如果线性规划问题有最优解，则必有一基本可行最优解。

上述定理的重要意义在于，它把一个最优化问题转化为一个组合问题，即在(8.2) ~ (8.5)式的 $m + n$ 个约束条件中，确定最优解应满足其中哪 n 个约束条件的问题。由此可知，只要对各种不同的组合进行测试，并比较每种情况下的目标函数值就能找到最优解。

盲目测试的计算量很大。Dantzig 于 1948 年首先提出了针对这一问题的单纯形算法。单纯形算法的特点是：

(1) 只对约束条件的若干组合进行测试，测试的每一步都使目标函数的值增加。

(2) 一般经过不大于 m 或 n 次迭代就可求得最优解。

自从提出单纯形算法后，人们已从实践经验中得到单纯形算法的性质(2)，但是直到 1982 年才由 Smale 给出其正确性的严格证明。

8.1.3 约束标准型线性规划问题的单纯形算法

当线性规划问题中没有不等式约束(8.2)和(8.4)式，而只有等式约束(8.3)式和变量非负约束(8.5)式时，称该线性规划问题具有标准形式。

为便于讨论，不妨先考察一类更特殊的标准形式线性规划问题。在这类线性规划问题中，在每个等式约束中至少有一个变量的系数为正，且这个变量只在该约束中出现。在每一约束方程中选择一个这样的变量，并以它作为变量求解该约束方程，这样选出来的变量称为左端变量或基本变量，其总数为 $m(= m_2)$ 个。剩下的 $n - m$ 个变量称为右端变量或非基本变量。这类特殊的标准形式线性规划问题称为约束标准型线性规划问题。

虽然约束标准型线性规划问题非常特殊，但是对于理解线性规划问题的单纯形算法非常重要。稍后将看到，任意一个线性规划问题可以转换为约束标准型线性规划问题。

先看一个约束标准型线性规划问题的例子：

$$\max z = -x_2 + 3x_3 - 2x_5 \tag{8.8}$$

$$\begin{aligned} \text{s.t.} \quad & x_1 + 3x_2 - x_3 + 2x_5 = 7 \\ & x_4 - 2x_2 + 4x_3 = 12 \\ & x_6 - 4x_2 + 3x_3 + 8x_5 = 10 \\ & x_i \geqslant 0 \ (i = 1,2,3,4,5,6) \end{aligned} \tag{8.9}$$

此例中，$n = 6$，$m = 3$；基本变量为 x_1，x_4 和 x_6；非基本变量为 x_2，x_3 和 x_5。注意，这里的目标函数(8.8)式中仅包含非基本变量。这实际上并不是一种特殊要求，因为出现在目标函数中的基本变量可以用约束方程代入消去。

对于任何约束标准型线性规划问题，只要将所有非基本变量都置为0，从约束方程式中解出满足约束的基本变量的值，即可求得一个基本可行解。当然，这个基本可行解未必是最优解。单纯形算法的基本思想就是，从一个基本可行解出发，进行一系列的基本可行解的变换。每次变换将一个非基本变量与一个基本变量互调位置，且保持当前的线性规划问题是一个与原问题完全等价的标准型线性规划问题。

为了便于表达，将(8.8)和(8.9)式所包含的信息记录在如图8-1所示的单纯形表中。

		x_2	x_3	x_5
z	0	-1	3	-2
x_1	7	3	-1	2
x_4	12	-2	4	0
x_6	10	-4	3	8

图8-1　单纯形表

该问题的一个明显的基本可行解是 $x = (7,0,0,12,0,10)$。

单纯形算法的第1步是选出使目标函数增加的非基本变量作为入基变量。查看单纯形表的第1行(也称之为 z 行)中标有非基本变量的各列中的值，依次让每一非基本变量从当前值开始增加，同时保持其余非基本变量仍为0；然后考察变化结果，看目标函数值是增加还是减小了。考察的目的是选出使目标函数增加的非基本变量作为入基变量。容易看出，z 行中的正系数非基本变量都满足要求。在上面单纯形表的 z 行中只有1列为正，即非基本变量 x_3 相应的列，其值为3。因此，选取非基本变量 x_3 作为入基变量。

单纯形算法的第2步是选取离基变量。在单纯形表中考察由第1步选出的入基变量所对应的列。在一个基本变量变为负值之前，查看入基变量可以增到多大。如果入基变量所在的列与基本变量所在行交叉处的表元素为负数，那么该元素将不受任何限制，相应的基本变量只会越变越大。如果入基变量所在列的所有元素都是负值，则目标函数无界，说明已经得到了问题的无界解。

如果选出的列中有一个或多个元素为正数，那么就要弄清到底是哪一个数首先限制了入

基变量值的增加。显然，这一受限的增加量可以用入基变量所在列的元素(称为主元素)来除主元素所在行的“常数列”(最左边的列)中元素而得到。所得到数值越小说明受到限制越多。因此，应该选取受到限制最多的基本变量作为离基变量，才能保证将入基变量与离基变量互调位置后，仍满足约束条件。

在上面的例子中，惟一的一个正值为 z 行元素 3，它所在列中有 2 个正元素，即 4 和 3。由于 $\min\{12/4, 10/3\} = 3$，故应该选取 x_4 为离基变量；入基变量 x_3 取值为 3。

单纯形算法的第 3 步是转轴变换。转轴变换的目的是将入基变量与离基变量互调位置。给入基变量一个增值，使之成为基本变量；同时修改离基变量，让入基变量所在列中离基变量所在行的元素值减为零，并使之成为非基本变量。

对上面的例子，首先解离基变量相应的方程

$$x_4 - 2x_2 + 4x_3 = 12$$

将入基变量 x_3 用离基变量 x_4 表示为

$$x_3 - \frac{1}{2}x_2 + \frac{1}{4}x_4 = 3$$

再将其代入其他基本变量 x_1 和 x_6 所在的行中消去 x_3，得到

$$x_1 + \frac{5}{2}x_2 + \frac{1}{4}x_4 + 2x_5 = 10$$

$$x_6 - \frac{5}{2}x_2 - \frac{3}{4}x_4 + 8x_5 = 1$$

代入目标函数得到

$$z = 9 + \frac{1}{2}x_2 - \frac{3}{4}x_4 - 2x_5$$

至此，可以形成新单纯形表如图 8-2 所示。

单纯形算法的第 4 步是转回并重复第 1 步，进一步改进目标函数值。

不断重复上述过程，直到 z 行的所有非基本变量系数都变成负值为止。这表明目标函数不可能再增加了。

在上面的单纯形表中，z 行元素惟一的正值为非基本变量 x_2 相应的列，其值为 1/2。因此，选取非基本变量 x_2 作为入基变量。它所在列中有惟一的正元素 5/2，即基本变量 x_1 相应行的元素。因此，选取 x_1 为离基变量。

再经步骤 3 的转轴变换得到的新单纯形表如图 8-3 所示。

		x_2	x_4	x_5
z	9	1/2	–3/4	2
x_1	10	5/2	1/4	2
x_3	3	–1/2	1/4	0
x_6	1	-5/2	–3/4	8

图 8-2　新单纯形表 1

		x_1	x	x_5
z	11	1/5	–.5	-12/5
x_2	4	5/2	1/10	4/5
x_3	5	1/5	3 0	2/5
x_6	11	1	–1/2	10

图 8-3　新单纯形表 2

新单纯形表 z 行的所有非基本变量系数都变成负值，因此求解过程结束。

整个问题的解可以从最后一张单纯形表的常数列中读出。在上面的单纯形表中可以看到，目标函数的最大值为 11；最优解为 $x^* = (0,4,5,0,0,11)$。

回顾单纯形算法的计算过程可以看出，整个过程可以用单纯形表的形式归纳为一系列基本矩阵运算。主要运算为转轴变换，该变换类似解线性方程组的高斯消去法中的消元变换。

不妨设当前的单纯形表如图 8-4 所示。

		x_{m+1}	x_{m+2}	…	x_n
z	c_0	c_{m+1}	c_{m+2}	…	c_n
x_1	b_1	a_{1m+1}	a_{1m+2}	…	a_{1n}
x_2	b_2	a_{2m+1}	a_{2m+2}	…	a_{2n}
⋮	⋮	⋮	⋮	…	⋮
x_m	b_m	a_{mm+1}	a_{mm+2}	…	a_{mn}

图 8-4　当前单纯形表

其中，$x_1, x_2, \cdots, x_m$ 为基本变量；$x_{m+1}, x_{m+2}, \cdots, x_n$ 为非基本变量。基本变量下标集为 $B = \{1, 2, \cdots, m\}$；非基本变量下标集为 $N = \{m+1, m+2, \cdots, n\}$；当前基本可行解为$(b_1, b_2, \cdots, b_m, 0, \cdots, 0)$。

单纯形算法计算步骤如下。

步骤 1　选入基变量。

如果所有 $c_j \leqslant 0$，则当前基本可行解为最优解，计算结束。否则，取 $c_e > 0$，相应的非基本变量 x_e 为入基变量。

步骤 2　选离基变量。

对于步骤 1 选出的入基变量 x_e，如果所有 $a_{ie} \leqslant 0$，$(i = 1, 2, \cdots, m)$，则最优解无界，计算结束。否则，计算

$$\theta = \min_{a_{ie} > 0}\left\{\frac{b_i}{a_{ie}}\right\} = \frac{b_k}{a_{ke}}$$

选取基本变量 x_k 为离基变量。

新的基本变量下标集为 $\overline{B} = B + \{e\} - \{k\}$；新的非基本变量下标集为 $\overline{N} = N + \{k\} - \{e\}$。

步骤 3　作转轴变换。

新单纯形表中各元素变换如下：

$$\begin{cases} \overline{b}_i = b_i - a_{ie}\dfrac{b_k}{a_{ke}} \ (i \in \overline{B}) \\ \overline{b}_e = \dfrac{b_k}{a_{ke}} \end{cases} \tag{8.10}$$

$$\begin{cases} \overline{a}_{ij} = a_{ij} - a_{ie}\dfrac{a_{kj}}{a_{ke}} \ (i \in \overline{B}, j \in \overline{N}) \\ \overline{a}_{ik} = -\dfrac{a_{ie}}{a_{ke}} \end{cases} \tag{8.11}$$

$$\begin{cases} \bar{a}_{ej} = \dfrac{a_{kj}}{a_{ke}} \ (j \in \overline{N}) \\ \bar{a}_{ek} = \dfrac{1}{a_{ke}} \end{cases} \tag{8.12}$$

$$\begin{cases} \bar{c}_i = c_i - c_e \dfrac{a_{ki}}{a_{ke}} \ (i \in \overline{N}) \\ c_k = -\dfrac{c_e}{a_{ke}} \end{cases} \tag{8.13}$$

步骤 4 转步骤 1。

8.1.4 将一般问题转化为约束标准型

有几种巧妙的办法可以将一般线性规划问题转化为约束标准型线性规划问题。

首先,需要把(8.2)或(8.4)式的不等式约束转化为等式约束。例如,(8.7)式中的不等式约束。具体做法是,引入松弛变量,利用松弛变量的非负性将不等式转化为等式。松驰变量记为 y_i,共有 $m_1 + m_3$ 个。在求解过程中,应当将松弛变量与原来变量 x_i 同样对待。在求解结束后,抛弃松弛变量。

例如,在引入松弛变量之后,目标函数(8.6)式未发生变化,而(8.7)式变换成如下形式:

$$\begin{aligned} & x_1 + 2x_3 + y_1 = 18 \\ & 2x_2 - 7x_4 + y_2 = 0 \\ & x_1 + x_2 + x_3 + x_4 = 9 \\ & x_2 - x_3 + 2x_4 - y_3 = 1 \end{aligned} \tag{8.14}$$

注意松弛变量前的符号由相应的原不等式的方向所决定。

为了进一步构造标准型约束,还需要引入 m 个人工变量,记为 z_i。

例如,在(8.14)式的每一等式约束中都引入一个人工变量,将其变换为

$$\begin{aligned} & z_1 - x_1 + 2x_3 + y_1 = 18 \\ & z_2 + 2x_2 - 7x_4 + y_2 = 0 \\ & z_3 + x_1 + x_2 + x_3 + x_4 = 9 \\ & z_4 + x_2 - x_3 + 2x_4 - y_3 = 1 \end{aligned} \tag{8.15}$$

至此,原问题已经变换为等价的约束标准型线性规划问题。

对极小化线性规划问题,只要将目标函数乘以 -1 即可化为等价的极大化线性规划问题。

8.1.5 一般线性规划问题的 2 阶段单纯形算法

细心的读者可能已经发现,除非所有 z_i 都是 0,否则(8.14)与(8.15)式并不等价。为了解决这个问题,在求解时必须分两个阶段进行。

第 1 阶段用一个辅助目标函数替代原来的目标函数(8.6)式,即

$$z' = -z_1 - z_2 - z_3 - z_4 = -(28 - 2x_1 - 4x_2 - 2x_3 + 4x_4 - y_1 - y_2 + y_3) \tag{8.16}$$

其中,后一个等式是将(8.15)式代入得到的。

这个线性规划问题称为原线性规划问题所相应的辅助线性规划问题。现在,对辅助线性规划问题用单纯形算法求解。显然,如果原线性规划问题有可行解,则辅助线性规划问题就有最

优解，且其最优值为 0，即所有 z_i 都为 0。在辅助线性规划问题最后的单纯形表中，所有 z_i 均为非基本变量。划掉所有 z_i 相应的列，剩下的就是只含 x_i 和 y_i 的约束标准型线性规划问题。换句话说，单纯形算法第 1 阶段的任务就是构造一个初始基本可行解。

单纯形算法第 2 阶段是求解由第 1 阶段导出的问题。此时要用原来的目标函数进行求解。如果在辅助线性规划问题最后的单纯形表中，z_i 不全为 0，则原线性规划问题没有可行解，从而原线性规划问题无解。

8.1.6 单纯形算法的描述和实现

下面讨论一般线性规划问题的 2 阶段单纯形算法的实现。我们用一个 C++ 类 LinearProgram 来表示解线性规划问题的单纯形算法。

```
class LinearProgram {
  public:
    LinearProgram(char * filename);
    ~LinearProgram();
    void solve();
  private:
    int enter(int objrow);
    int leave(int col);
    int simplex(int objrow);
    int phase1();
    int phase2();
    int compute();
    void swapbasic(int row, int col);
    void pivot(int row, int col);
    void stats();
    void setbasic(int * basicp);
    void output();
    int m,          // 约束总数
        n,          // 变量数
        m1,         // 不等式约束数(≤)
        m2,         // 等式约束数
        m3,         // 不等式约束数(≥)
        n1, n2,     //  n1 = n + m3; n2 = n1 + m1;
        error,      // 记录错误类型
        * basic,    // 基本变量下标
        * nonbasic; // 非基本变量下标
    double * * a, minmax;
};
```

其中，主要数据项是存储单纯形表的二维数组 a，其存储内容如图 8-5 所示。实际存储的是粗线框表示的部分。

0	a_{01} … a_{0n}	0			
b_1 ⋮ b_{m1}	a_{11} … a_{1n} ⋮ ⋮ a_{m11} … a_{m1n}	0	1 ⋱ 1		
b_{m1+1} ⋮ b_{m1+m2}	a_{m1+11} … a_{m1+1n} ⋮ ⋮ a_{m1+m21} … a_{m1+m2n}	0		1 ⋱ 1	
$b_{m1+m2+1}$ ⋮ b_m	$a_{m1+m2+11}$ … $a_{m1+m2+1n}$ ⋮ ⋮ a_{m1} … a_{mn}	−1 ⋱ −1			1 ⋱ 1
0	$\sum_{i=m1+1}^{m} a_{i1}$ … $\sum_{i=m1+1}^{m} a_{in}$	−1 … −1	0 … 0	0 … 0	0 … 0
0	0 … 0	0 … 0	0 … 0	-1 … -1	−1 … −1

图 8-5 初始单纯形表

1. 构造初始单纯形表

首先,从标准输入文件中读入数据,构造初始单纯形表。

```
LinearProgram::LinearProgram(char * filename)
{
    ifstream inFile;
    int i,j;
    double value;
    cout<<"按照下列格式输入数据:"<<endl;
    cout<<"1 : +1 (max) 或 -1 (min); m; n"<<endl;
    cout<<"2 : m1; m2; m3"<<endl;
    cout<<"约束系数和右端项"<<endl;
    cout<<"目标函数系数"<<endl<<endl;
    error=0;
    inFile.open(filename);
    inFile>>minmax;
    inFile>>m;
    inFile>>n;
    // 输入各类约束数
    inFile>>m1;
    inFile>>m2;
    inFile>>m3;
    if ( m != m1+m3+m2 ) error=1;
    n1 = n + m3;
    n2 = n + m1 + m3;
    Make2DArray(a,m+2,n1+1);
    basic = new int[m+2];
```

```
        nonbasic = new int[n1 + 1];
        // 初始化基本变量和非基本变量
        for (i=0;i<=m+1;i++)
            for (j=0;j<=n1;j++) a[i][j]=0.0;
        for (j=0;j<=n1;j++) nonbasic[j] = j;
        // 引入松弛变量和人工变量
        for (i=1, j=n1+1; i<=m; i++,j++) basic[i] = j;
        for (i=m-m3+1,j=n+1;i<=m;i++,j++){
            a[i][j] = -1.0;
            a[m+1][j] = -1.0;
        }
        // 输入约束系数和右端项
        for (i=1;i<=m;i++){
            for (j=1;j<=n;j++){
                inFile>>value;
                a[i][j] = value;
            }
            inFile>>value;
            if (value<0) error=1;
            a[i][0] = value;
        }
        // 输入目标函数系数
        for (j=1;j<=n;j++) {
            inFile>>value;
            a[0][j] = value * minmax;
        }
        // 引入人工变量,构造第1阶段的辅助目标函数
        for (j=1;j<=n;j++){
            for (i=m1+1, value=0.0; i<=m;i++) value += a[i][j];
            a[m+1][j] = value;
        }
        inFile.close();
    }
```

2. 约束标准型线性规划问题的单纯形算法

函数 simplex(objrow)根据目标函数系数所在的行 objrow,执行约束标准型线性规划问题的单纯形算法。

```
    int LinearProgram::simplex(int objrow)
    {
        for(int row=0;;){
            int col=enter(objrow);
            if ( col>0 ) row=leave(col);
            else return 0;
```

```
            if ( row > 0 ) pivot(row,col);
            else return 2;
        }
    }
```

其中,函数 enter(objrow)根据目标函数系数所在的行 objrow,选取入基变量。

```
int LinearProgram::enter(int objrow)
{
    double temp = DBL_EPSILON;
    for (int j = 1,col = 0; j< = n1; j+ + ){
        if ( nonbasic[j] < = n2 && a[objrow][j] > temp ){
            col = j; temp = a[objrow][j];
//          break;                       // Bland 避免循环法则
        }
    }
    return col;
}
```

函数 leave(col)根据入基变量所在的列 col,选取离基变量。

```
int LinearProgram::leave(int col)
{
    double temp = DBL_MAX;
    for (int i = 1, row = 0; i< = m; i+ + ){
        double val = a[i][col];
            if ( val > DBL_EPSILON ) {
                val = a[i][0]/val;
                if ( val < temp ){
                    row = i; temp = val;
                }
            }
    }
    return row;
```

函数 pivot(row, col)以入基变量所在的列 col 和离基变量所在行 row 交叉处元素 a[row][col]为轴心,作转轴变换。

```
void LinearProgram::pivot(int row, int col)
{
    for (int j = 0; j< = n1; j+ + )
        if ( j ! = col ) a[row][j] = a[row][j]/a[row][col];
    a[row][col] = 1.0/a[row][col];
```

```
            for (int i=0; i<=m+1; i++)
                if ( i != row ){
                    for (int j=0; j<=n1; j++)
                        if ( j != col ){
                            a[i][j]=a[i][j]-a[i][col]*a[row][j];
                            if ( fabs(a[i][j])<DBL_EPSILON) a[i][j]=0.0;
                        }
                    a[i][col] = -a[i][col]*a[row][col];
                }
            swapbasic(row,col);
        }
```

函数 swapbasic(row, col)交换基本变量 row 和非基本变量 col 的位置。

```
        void LinearProgram::swapbasic(int row, int col)
        {
            int temp=basic[row];
            basic[row]=nonbasic[col];
            nonbasic[col]=temp;
        }
```

3.2 阶段单纯形算法

函数 compute()对一般的线性规划问题执行 2 阶段单纯形算法。

```
        int LinearProgram::compute()
        {
            if ( error>0 ) return error;
            if ( m != m1 ){
                error=phase1();
                if (error>0) return error;
            }
            return phase2();
        }
```

其中,构造初始基本可行解的第 1 阶段单纯形算法由 phase1()实现。辅助目标函数存储在数组 a 的第 trows 行。

```
        int LinearProgram::phase1()
        {
            error=simplex(m+1);
            if ( error>0 ) return error;
            for (int i=1;i<=m;i++)
                if ( basic[i]>n2 ){
```

```
            if ( a[i][0] > DBL_EPSILON ) return 3;
            for (int j = 1;j < = n1;j + + )
                if ( fabs(a[i][j]) > = DBL_EPSILON ){
                    pivot (i,j);
                    break;

        }
    return 0;
    }
```

单纯形算法第 2 阶段根据第 1 阶段找到的基本可行解,对原来的目标函数用单纯形算法求解。原目标函数存储在数组 a 的第 0 行。

```
int LinearProgram::phase2()
{
    return simplex(0);
}
```

函数 solve()是执行 2 阶段单纯形算法的公有函数。

```
void LinearProgram::solve()
{
    cout< <endl< <"* * *  线性规划 - - - 单纯形算法  * * *"< <endl< <endl;
    error = compute();
    switch (error){
        case 0: output(); break;
        case 1: cout< <"-输入数据错误 - -"< <endl; break;
        case 2: cout< <"-无界解 - -"< <endl; break;
        case 3: cout< <"-无可行解 - -"< <endl;
    }
    cout< <"计算结束"< <endl;
}
```

函数 output()输出 2 阶段单纯形算法的计算结果。

```
void LinearProgram::output()
{
        int width = 8, * basicp;
        double zero = 0.0;
        basicp = new int[n + m + 1];
        setbasic(basicp);
        cout.setf(ios::fixed|ios::showpoint|ios::right);
        cout.precision(4);
```

```
    stats();
    cout<<endl<<"最优值:"<<-minmax*a[0][0]<<endl<<endl;
    cout<<"最优解:"<<endl<<endl;
    for (int j=1;j<=n;j++){
        cout<<"x"<<j<<"=";
        if (basicp[j]!=0) cout<<setw(width)<<a[basicp[j]][0];
        else cout<<setw(width)<<zero;
        cout<<endl;
    }
    cout<<endl;
    delete []basicp;
}
```

8.1.7 退化情形的处理

用单纯形算法解一般的线性规划问题时,可能会遇到退化的情形,即在迭代计算的某一步中,常数列中的某个元素的值变成0,使得相应的基本变量取值为0。如果选取退化的基本变量为离基变量,则作转轴变换前后的目标函数值不变。在这种情况下,算法不能保证目标函数值严格递增,因此可能出现无限循环。

考察下面的由 Beale 在 1955 年提出的退化问题的例子。

$$\max z = \frac{3}{4}x_1 - 20x_2 + \frac{1}{2}x_3 - 6x_4$$

$$\text{s.t.} \quad \frac{1}{4}x_1 - 8x_2 - x_3 + 9x_4 \leqslant 0$$

$$\frac{1}{2}x_1 - 12x_2 - \frac{1}{2}x_3 + 3x_4 \leqslant 0$$

$$x_3 \leqslant 1$$

$$x_i \geqslant 0 \ (i = 1,2,3,4)$$

按照2阶段单纯形算法求解该问题将出现无限循环。

Bland 提出的用单纯形算法解退化的线性规划问题时,避免循环是一个简单易行的方法。Bland 提出在单纯形算法迭代中,按照下面的两个简单规则来避免循环。

规则 1 设 $e = \min\{j \mid c_j > 0\}$,取 x_e 为入基变量。

规则 2 设 $k = \min\left\{ l \mid \frac{b_l}{a_{le}} = \min_{a_{ie}>0}\left\{\frac{b_i}{a_{ie}}\right\}\right\}$,取 x_k 为离基变量。

前面的算法 leave(col) 已经按照规则2选取离基变量。选取入基变量的算法 enter(objrow) 中只要增加一个 break 语句即可。参见前面的算法描述。

8.1.8 应用举例

1. 仓库租赁问题

某企业计划为流通的货物租赁若干批仓库。要求:必须保证在时间段 $i = 1,2,\cdots,n$,有 b_i

的仓库容量可用。现有若干仓库源可供选择。设 c_{ij} 是从时间段 i 到时间段 j 租用 1 个单位仓库容量的价格，$1 \leqslant i \leqslant j \leqslant n$。问题是：应如何安排仓库租赁计划才能满足各时间段的仓库需求，且使租赁费用最少。

设租用时间段 i 到时间段 j 的仓库容量为 y_{ij}，$1 \leqslant i \leqslant j \leqslant n$，则租用仓库的总费用为

$$\sum_{i=1}^{n}\sum_{j=i}^{n} c_{ij} y_{ij}$$

在时间段 k 可用的仓库容量为

$$\sum_{i=1}^{k}\sum_{j=k}^{n} y_{ij}$$

由此可见，仓库租赁问题可表述为下面的线性规划问题：

$$\min \sum_{i=1}^{n}\sum_{j=i}^{n} c_{ij} y_{ij}$$

$$\text{s.t.} \quad \sum_{i=1}^{k}\sum_{j=k}^{n} y_{ij} \geqslant b_k \quad (k = 1,2,\cdots,n)$$

$$y_{ij} \geqslant 0 \ (1 \leqslant i \leqslant j \leqslant n)$$

设 $m = n(n+1)/2$

$$(y_{11}, y_{12}, \cdots, y_{1n}, y_{22}, y_{23}, \cdots, y_{2n}, \cdots, y_{nn}) = (x_1, x_2, \cdots, x_m)$$

$$(c_{11}, c_{12}, \cdots, c_{1n}, c_{22}, \cdots, c_{23}, \cdots, c_{2n}, \cdots, c_{nn}) = (d_1, d_2, \cdots, d_m)$$

上述线性规划问题可表述为 n 个约束和 m 个变量的标准型线性规划问题：

$$\min d^{\mathrm{T}} x$$

$$\text{s.t.} \quad \boldsymbol{A} x \geqslant b$$

$$x \geqslant 0$$

其中，
$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 1 & 1 & 1 & 1 & \cdots & 1 & 0 & & & \\ 0 & 0 & 1 & \cdots & 1 & 0 & 1 & \cdots & 1 & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & 0 & 0 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

下面的算法从给定的仓库租赁问题输入构造矩阵 $\boldsymbol{A}$ 和相应的线性规划问题的输入参数，从而可用前面讨论的单纯形算法求解。

```
void input(char * filename,char * file)
{
    ifstream inFile;
    ofstream outFile;
    int * b, * c, * * d;
    int i,j,k,p,q,n,m;
    inFile.open(filename);
    inFile> > n;
    m = n * (n + 1)/2;
    b = new int [n];
    c = new int [m];
    Make2DArray(d,n,m);
```

```
    for(i=0;i<n;i++) inFile>>b[i];
    for(i=0,k=0;i<n;i++)
        for(j=0;j<n-i;j++)
            inFile>>c[k++];
    inFile.close();
    for(i=0;i<n;i++)
        for(j=0;j<m;j++) d[i][j]=0;
    for(k=0;k<n;k++){
        p=n-k;q=k*n-k*(k-1)/2;
        for(i=0;i<p;i++)
          for(j=i;j<p;j++)
            d[i+k][q+j]=1;
    }
    outFile.open(file);
    outFile<<-1<<" "<<n<<" "<<m<<endl;
    outFile<<0<<" "<<0<<" "<<n<<endl;
    for(i=0;i<n;i++){
        for(j=0;j<m;j++)
            outFile<<d[i][j]<<" ";
        outFile<<b[i]<<endl;
    }
    for(i=0;i<m;i++) outFile<<c[i]<<" ";
    outFile<<endl;
    outFile.close();
    delete []b; delete []c;
    Delete2DArray(d,n);
}
```

8.2 最大网络流问题

8.2.1 网络与流

1. 基本概念和术语

先介绍与网络流有关的一些基本概念。

(1) 网络

设 G 是一个简单有向图，$G=(V,E)$，$V=\{1,2,\cdots,n\}$。在 V 中指定一个顶点 s，称为源；指定另一个顶点 t，称为汇。对于有向图 G 的每一条边 $(v,w)\in E$，对应有一个值 $\text{cap}(v,w)\geqslant 0$，称它为边的容量。通常把这样的有向图 G 称为一个网络。

(2) 网络流

网络上的流是定义在网络的边集合 E 上的一个非负函数 $\text{flow}=\{\text{flow}(v,w)\}$，并称 $\text{flow}(v,w)$ 为边 (v,w) 上的流量。

(3) 可行流

满足下述条件的流 flow 称为可行流：

① 容量约束　对每一条边$(v,w)\in E, 0\leqslant \mathrm{flow}(v,w)\leqslant \mathrm{cap}(v,w)$。

② 平衡约束　对于中间顶点，流出量 = 流入量。即对每个 $v\in V(v\neq s,t)$ 有

$$顶点\ v\ 的流出量 - 顶点\ v\ 的流入量 = 0$$

即

$$\sum_{(v,w)\in E}\mathrm{flow}(v,w)-\sum_{(w,v)\in E}\mathrm{flow}(w,v)=0$$

对于源 s

$$s\ 的流出量 - s\ 的流入量 = 源的净输出量\ f$$

即

$$\sum_{(s,v)\in E}\mathrm{flow}(s,v)-\sum_{(v,s)\in E}\mathrm{flow}(v,s)=f$$

对于汇 t

$$t\ 的流入量 - t\ 的流出量 = 汇的净输入量\ f$$

即

$$\sum_{(v,t)\in E}\mathrm{flow}(v,t)-\sum_{(t,v)\in E}\mathrm{flow}(t,v)=f$$

式中，f 称为这个可行流的流量，即源的净输出量(或汇的净输入量)。

可行流总是存在的。例如，让所有边的流量 $\mathrm{flow}(v,w)=0$，就得到一个流量 $f=0$ 的可行流(称为 0 流)。

(4) 边流

对于网络 G 的一个给定的可行流 flow，将网络中满足 $\mathrm{flow}(v,w)=\mathrm{cap}(v,w)$ 的边称为饱和边；称 $\mathrm{flow}(v,w)<\mathrm{cap}(v,w)$ 的边为非饱和边；称 $\mathrm{flow}(v,w)=0$ 的边为零流边；称 $\mathrm{flow}(v,w)>0$ 的边为非零流边。当边(v,w)既不是一条零流边也不是一条饱和边时，称为弱流边。

(5) 最大流

最大流问题即求网络 G 的一个可行流 flow，使其流量 f 达到最大。即 flow 满足

$$0\leqslant \mathrm{flow}(v,w)\leqslant \mathrm{cap}(v,w),(v,w)\in E$$

且

$$\sum\mathrm{flow}(v,w)-\sum\mathrm{flow}(w,v)=\begin{cases}f & v=s\\ 0 & v\neq s,t\\ -f & v=t\end{cases}$$

(6) 流的费用

在实际应用中，与网络流有关的问题不仅涉及流量，而且还有费用因素。此时，网络的每一条边(v,w)除了给定容量 $\mathrm{cap}(v,w)$ 外，还定义了一个单位流量费用 $\mathrm{cost}(v,w)$。对于网络中一个给定的流 flow，其费用定义为

$$\mathrm{cost}(\mathrm{flow})=\sum_{(v,w)\in E}\mathrm{cost}(v,w)\times\mathrm{flow}(v,w)$$

(7) 残流网络

对于给定的一个网络 G 及其上的一个流 flow，网络 G 关于流 flow 的残流网络 G^* 与 G 有相同的顶点集 V，而网络 G 中的每一条边对应于 G^* 中的 1 条边或两条边。设(v,w)是 G 的一条边。当 $\mathrm{flow}(v,w)>0$ 时，(w,v) 是 G^* 中的一条边，该边的容量为 $\mathrm{cap}^*(w,v)=\mathrm{flow}(v,w)$；当 $\mathrm{flow}(v,w)<\mathrm{cap}(v,w)$ 时，(v,w) 是 G^* 中的一条边，该边的容量为 $\mathrm{cap}^*(v,w)=\mathrm{cap}(v,w)-\mathrm{flow}(v,w)$。

按照残流网络的定义，当原网络 G 中的边(v,w)是一条零流边时，残流网络 G^* 中有惟

一的一条边(v , w)与之对应,且该边的容量为 cap(v , w)。当原网络 G 中的边(v , w)是一条饱和边时,残流网络 G^* 中有惟一的一条边(w , v)与之对应,该边的容量为 cap(v , w)。当原网络 G 中的边(v , w)是一条弱流边时,残流网络 G^* 中有两条边(v , w)和(w , v)与之对应,这两条边的容量分别为 cap(v , w) - flow(v , w)和 flow(v , w)。

残流网络是设计与网络流有关算法的重要工具。

2. 流网络数据结构

以下用类 EDGE 表示网络中的边。

```
class EDGE
{
   int pv, pw, pcap, pcost, pflow;
   public:
     EDGE(int v, int w, int cap,int cost) :
       pv(v), pw(w), pcap(cap), pcost(cost), pflow(0) { }
     int v() const { return pv; }
     int w() const { return pw; }
     int cap() const { return pcap; }
     int cost() const { return pcost; }
     int wt(int v) const { return from(v) ? -pcost : pcost; }
     int flow() const { return pflow; }
     bool from (int v) const
     {
          return pv = = v;
     }
     bool residual(int v) const
     {
          return (pv = = v && pcap - pflow > 0 || pw = = v && pflow > 0 );
     }
     int other(int v) const
     {
          return from(v) ? pw : pv;
     }
     int capRto(int v) const
     {
          return from(v) ? pflow : pcap - pflow;
     }
     int costRto(int v) const
     {
          return from(v) ? -pcost : pcost;
     }
     void addflowRto(int v, int d)
     {
```

```
            pflow += from(v) ? -d : d;
        }
    };
```

其中,私有成员 pv 和 pw 分别表示边的起点和终点;pcap, pcost 和 pflow 分别表示边的容量、费用和流量。

类 EDGE 的大部分公有成员是简单自明的,有些在用到时再进一步解释。

函数 from 和 other 与有向边的方向有关。如果 e 是指向一条边的指针. e - > from(v)返回 true 时, v 是边 e 的起点; e - > other(v)则返回边 e 的不同于 v 的另一个端点。

函数 residual, capRto, costRto, addflowRto 与残流网络有关。

函数 residual(v)用于判断残流网络中是否有一条以 v 为起点的边。

函数 capRto 给出残流网络中边的容量。如果 e 是指向边(v, w)的指针, e 的容量为 c, 流量为 f, 则按残流网络的定义 e - > capRto(w)是 $c-f$, 而 e - > capRto(v)是 f。

函数 costRto 给出残流网络中边的费用。如果边 e 的费用是 cost, 则按残流网络的定义 e - > costRto(w)是 cost, 而 e - > capRto(v)是 - cost。

函数 addflowRto 改变残流网络中边的流量。如果 e 是指向边(v, w)的指针, e 的流量为 f, 则 e - > addflowRto(w, d)将 e 的流量改变为 $f+d$, 而 e - > addflowRto(v, d)将 e 的流量改变为 $f-d$。

下面用类 GRAPH 表示一般的网络。

```
template <class Edge> class GRAPH
{
    int Vcnt, Ecnt; bool digraph;
    vector< vector<Edge *> > adj;
    public:
    GRAPH(int V, bool digraph = false) :
        adj(V+1), Vcnt(V+1), Ecnt(0), digraph(digraph)
    {
        for (int i = 0; i <= V; i++)
          adj[i].assign(V+1, 0);
    }

    GRAPH(){}

    int V() const { return Vcnt; }
    int E() const { return Ecnt; }
    bool directed() const { return digraph; }
    void insert(Edge *e)
    {
        int v = e->v(), w = e->w();
        if (adj[v][w] == 0) Ecnt++;
        adj[v][w] = e;
```

```
        if (! digraph) adj[w][v] = e;
    }
    void remove(Edge * e)
    {
        int v = e->v(), w = e->w();
        if (adj[v][w] != 0) Ecnt--;
        adj[v][w] = 0;
        if (! digraph) adj[w][v] = 0;
    }
    Edge * edge(int v, int w) const
    {
        return adj[v][w];
    }
    void read(char *filename,int& s,int& t,int &se,int &te)
    {
        int i,j, n,m,nmax,cap,cost;
        ifstream inFile;
        inFile.open(filename);
        inFile>> n >> m >> s >> se>> t >>te>>nmax;
        for (int k=0;k<m;k++){
          inFile>> i >> j >> cap >> cap >> cost;
          if (cap>0) insert(new EDGE(i, j, cap, cost));
        }
        insert(new EDGE(nmax+1,0, se, 0));
        s=nmax+1;
        inFile.close();
    }
    void checksd(int s,int t, int &ss, int &dd)
    {
        ss=0;dd=0;
        for (int i=0;i<Vcnt;i++){
          if (adj[s][i] && adj[s][i]->from(s) && adj[s][i]->flow()>0 )
            ss+=adj[s][i]->flow();
          if (adj[i][t] && adj[i][t]->from(i) && adj[i][t]->flow()>0 )
            dd+=adj[i][t]->flow();
        }
    }
};
```

上述结构用图的邻接矩阵表示一般网络,其私有成员 Vcnt 和 Ecnt 分别表示网络中的顶点数和边数;adj 是邻接矩阵;digraph 是有向图标志。

类 GRAPH 的大部分公有成员是简单自明的,有些在用到时再进一步解释。

网络的搜索游标功能是有序地搜索网络中与某个顶点相关联的各条边,我们将其定义为

一个特殊的类 adjIterator。

```
template <class Edge>
class adjIterator
{
    const GRAPH<Edge> &G;
    int i, j, v;
    public:
      adjIterator(const GRAPH<Edge> &G, int v) : G(G), v(v), i(0), j(0) { }
      Edge * beg()
      {
        i = -1; j= -1 ; return nxt();
      }
      Edge * nxt()
      {
        for (i++; i < G.V(); i++)
           if (G.edge(v, i)) return G.edge(v, i);
        for (j++; j < G.V(); j++)
           if (G.edge(j, v)) return G.edge(j, v);
        return 0;
      }
      bool end() const
      {
        return (i >= G.V() && j >= G.V());
    }
};
```

其中,beg()是搜索的起始边;nxt()是下一条要搜索的边;end()表示搜索结束。

8.2.2 增广路算法

1. 算法基本思想

设 P 是网络 G 中联结源 s 和汇 t 的一条路。定义路的方向是从 s 到 t。可以将路 P 上的边分成 2 类:一类边的方向与路的方向一致,称为向前边,其全体记为 P^+;另一类边的方向与路的方向相反,称为向后边,其全体记为 P^-。

设 flow 是一个可行流,P 是从 s 到 t 的一条路,若 P 满足下列条件:

(1) 在 P 的所有向前边(v,w)上,$flow(v,w) < cap(v,w)$,即 P^+ 中的每一条边都是非饱和边;

(2) 在 P 的所有向后边(v,w)上,$flow(v,w) > 0$,即 P^- 中的每一条边都是非零流边;

则称 P 为关于可行流 flow 的一条可增广路。

可增广路是残流网络中一条容量大于 0 的路。

将具有上述特征的路 P 称为可增广路,是因为可以通过修正路 P 上所有边流量 $flow(v,w)$,将当前可行流改进成一个流值更大的可行流。

具体做法是：

(1) 使不属于可增广路 P 的边(v,w)上的流量保持不变。

(2) 可增广路 P 上的所有边(v,w)上的流量按下述规则变化：

· 在向前边(v,w)上，$\text{flow}(v,w)+d$。

· 在向后边(v,w)上，$\text{flow}(v,w)-d$。

也就是按下面的公式修改当前的流：

$$\text{flow}(v,w)=\begin{cases}\text{flow}(v,w)+d & (v,w)\in P^{+}\\ \text{flow}(v,w)-d & (v,w)\in P^{-}\\ \text{flow}(v,w) & (v,w)\notin P\end{cases}$$

其中，d 称为可增广量。它按下述原则确定：d 取得尽量大，使变化后的流仍为可行流。不难看出，按照这个原则，d 既不能超过每条向前边(v,w)的 $\text{cap}(v,w)-\text{flow}(v,w)$，也不能超过每条向后边$(v,w)$的 $\text{flow}(v,w)$。因此 d 应该等于向前边上的 $\text{cap}(v,w)-\text{flow}(v,w)$ 与向后边上的 $\text{flow}(v,w)$ 的最小值，也就是残流网络中 P 的最大容量。

增广路定理 设 flow 是网络 G 的一个可行流，如果不存在从 s 到 t 关于 flow 的可增广路 P，则 flow 是 G 的一个最大流。

2. 算法描述

根据前面的讨论，可设计求最大流的增广路算法如下。该算法也常称为 Ford Fulkerson 算法。

```
template <class Graph, class Edge> class MAXFLOW
{
    const Graph &G;
    int s, t, maxf;
    vector<int> wt;
    vector<Edge *> st;
    int ST(int v) const { return st[v]->other(v); }

    void augment(int s, int t)
    {
        int d = st[t]->capRto(t);
        for (int v = ST(t); v != s; v = ST(v))
          if (st[v]->capRto(v) < d)  d = st[v]->capRto(v);
        st[t]->addflowRto(t, d);
        maxf += d;
        for ( v = ST(t); v != s; v = ST(v))
          st[v]->addflowRto(v, d);
    }

    bool pfs();

    public:
```

```
    MAXFLOW(const Graph &G, int s, int t,int &maxflow) :
        G(G), s(s), t(t), st(G.V()), wt(G.V()),maxf(0)
    {
        while (pfs()) augment(s, t);
        maxflow + = maxf;
    }
};
```

上面描述的是一个适用于一大类算法的广义的算法框架。算法的基本思想是,用一个PFS(优先级优先搜索,Priority First Search)搜索算法找到网络中的一条从 s 到 t 的可增广路,然后沿此可增广路增流,直到网络中找不到可增广路时为止。

算法中用向量 st 来存储 pfs 搜索到的网络支撑树,用 st[v]存储指向树边 e 的指针,e 是连接 v 和 v 的父结点的边。函数 ST[v]返回支撑树中 v 的父结点。

算法 augment(s,t)首先沿 st 给出的可增广路计算可增广量 d,然后再沿可增广路增流。

整个算法的关键和难点是如何寻找关于当前可行流的可增广路,特别是当网络中顶点数和边数较多时,寻找可增广路的算法 pfs 的效率至关重要。

```
template <class Graph, class Edge>
bool MAXFLOW<Graph, Edge>::pfs()
{
    PQ<int> pQ(G.V(), wt);
    for (int v = 0; v < G.V(); v++)
    { wt[v] = 0; st[v] = 0; pQ.insert(v); }
    wt[s] = M; pQ.change(s);
    while (! pQ.empty())
    {
        int v = pQ.deletemax(); wt[v] = M;
        if (v == t || (v ! = s && st[v] == 0)) break;
        adjIterator<Edge> A(G, v);
        for (Edge * e = A.beg(); ! A.end(); e = A.nxt())
        {
            int w = e->other(v);
            int cap = e->capRto(w);
            int P = cap < wt[v] ? cap : wt[v];
            if (cap > 0 && P > wt[w])
            { wt[w] = P; pQ.change(w); st[w] = e; }
        }
    }
    return st[t] ! = 0;
}
```

上面描述的算法 pfs 实际上是一个适用于寻找可增广路的算法框架。算法中用一个优先队列 pQ 记录搜索优先级。按搜索优先级依次搜索网络的各条边,直至找到一条可增广路。

不同的优先级将导致不同的搜索效果。例如，上面的算法 pfs 中将优先级定义为残流网络中边的容量。相应的算法也称为最大容量增广路算法。如果用最短路长作为优先级，则算法中的优先队列就是一个普通队列。此时的优先级搜索就是广度优先搜索，与此相应的增广路算法称为最短增广路算法。

算法中的 M 是网络最大边容量的一个上界。向量 wt 用于记录搜索顶点的优先级。优先队列类 PQ 可用堆实现。

```
template <class keyType> class PQ
{
    int d, N;
    vector<int> pq, qp;
    const vector<keyType> &a;
    void exch(int i, int j)
    {
        int t = pq[i]; pq[i] = pq[j]; pq[j] = t;
        qp[pq[i]] = i; qp[pq[j]] = j;
    }
    void fixUp(int k)
    {
        while (k > 1 && a[pq[(k+d-2)/d]] < a[pq[k]])
        {
            exch(k, (k+d-2)/d); k = (k+d-2)/d;
        }
    }
    void fixDown(int k, int N)
    {
        int j;
        while ((j = d*(k-1)+2) <= N)
        {
            for (int i = j+1; i < j+d && i <= N; i++)
                if (a[pq[j]] < a[pq[i]]) j = i;
                if (!(a[pq[k]] < a[pq[j]])) break;
                exch(k, j); k = j;
        }
    }
  public:
  PQ(int N, const vector<keyType> &a, int d = 3) :
    a(a), pq(N+1, 0), qp(N+1, 0), N(0), d(d) { }
  int empty() const { return N == 0; }
  void insert(int v) { pq[++N] = v; qp[v] = N; fixUp(N); }
  int deletemax() { exch(1, N); fixDown(1, N-1); return pq[N--]; }
  void change(int k) { fixUp(qp[k]); }
};
```

3. 算法的计算复杂性

容易看出,增广路算法的效率由下面两个因素所确定:

(1) 整个算法找可增广路的次数。

(2) 每次找可增广路所需的时间。

由此可见,求网络最大流的增广路算法的效率主要由算法 pfs 确定。而 pfs 算法的效率又与优先队列的选择密切相关。当优先队列以最大容量为优先级时,得到最大容量增广路算法。当优先队列以最短路为优先级时,得到最短增广路算法。

如果给定的网络中有 n 个顶点和 m 条边,且每条边的容量不超过 M。可以证明,在一般情况下,增广路算法中找可增广路的次数不超过 nM 次。

对于最短增广路算法,在最坏情况下,找可增广路的次数不超过 $nm/2$ 次。找 1 次可增广路最多需要 $O(m)$ 计算时间。因此,在最坏情况下,最短增广路算法所需的计算时间为 $O(nm^2)$。当给定的网络是稀疏网络时,即 $m = O(n)$ 时,最短增广路算法所需的计算时间为 $O(n^3)$。

对于最大容量增广路算法,在最坏情况下,找可增广路的次数不超过 $2m\log M$ 次。由于使用堆来存储优先队列,找 1 次增广路最多需要 $O(n\log n)$ 计算时间。因此,在最坏情况下,最大容量增广路算法所需的计算时间为 $O(mn\log n\log M)$。当给定的网络是稀疏网络时,最大容量增广路算法所需的计算时间为 $O(n^2\log n\log M)$。

8.2.3 预流推进算法

1. 算法基本思想

增广路算法的特点是找到可增广路后,立即沿可增广路对网络流进行增广。每一次增广可能需要对最多 $n-1$ 条边进行操作。因此,在最坏情况下,每一次增广需要 $O(n)$ 计算时间。在有些情况下,这个代价是很高的。下面是一个极端的例子。

在图 8-6 所示的网络中,$s=1$,$t=20$。边(1,2),(2,3),…,(8,9)上的容量为 10,其他边(顶点 9 的出边和顶点 20 的入边)的容量均为 1。无论用哪种增广路算法,都会找到 10 条增广路,每条路长为 10,容量为 1。因此,总共需要 10 次增广,每次增广需要对 10 条边进行操作,每条边增广 1 个单位流量。然而,注意到这 10 条增广路中的前 9 个顶点(前 8 条边)是完全一样的。如果直接将前 8 条边的流量增广 10 个单位,而只对后面长为 2 的不同的有向路单独操作,就可以节省许多计算时间。这就是预流推进(preflow push)算法的基本思想。也就是说,预流推进算法注重对每一条边的增流,而不必每次一定对一条增广路增流。通常将沿一条边增流的运算称为一次推进(push)。

在算法的推进过程中,网络流满足容量约束,但一般不满足流量平衡约束。此外,从每个顶点(s 和 t 除外)流出的流量之和总是小于等于流入该顶点的流量之和。这种流称为预流(preflow),这也是这类算法被称为预流推进算法的原因。下面先给出预流的严格定义。

给定网络 $G=(V,E)$,一个预流是定义在 G 的边集 E 上的一个正边流函数。该函数满足容量约束,即对 G 的每一条边 $(v,w)\in E$,满足 $0\leqslant \text{flow}(v,w)\leqslant \text{cap}(v,w)$。

对 G 的每一中间顶点满足:流出量小于或等于流入量。即对每个 $v\in V(v\neq s,t)$有

$$\sum_{(v,w)\in E}\text{flow}(v,w)\leqslant \sum_{(w,v)\in E}\text{flow}(w,v)$$

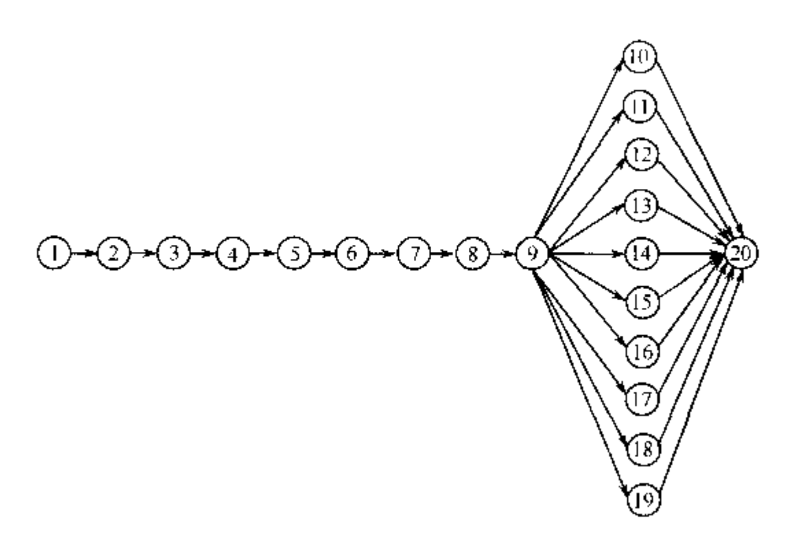


图 8-6　说明增广路算法的例子

满足条件

$$\sum_{(v,w)\in E} \text{flow}(v,w) < \sum_{(w,v)\in E} \text{flow}(w,v)$$

的中间顶点 v 称为活顶点。量

$$\sum_{(w,v)\in E} \text{flow}(w,v) - \sum_{(v,w)\in E} \text{flow}(v,w)$$

称为顶点 v 的存流。按此定义,源 s 和汇 t 不可能成为活顶点。

对网络 G 上的一个预流,如果存在活顶点,则说明该预流不是可行流。预流推进算法就是要选择活顶点,并通过把一定的流量推进到它的邻点,尽可能地将当前活顶点处正的存流减少为 0,直至网络中不再有活顶点,从而使预流成为可行流。如果当前活顶点有多个邻点,那么首先推进到哪个邻点呢?由于算法最后的目的是尽可能将流推进到汇点 t,因此算法应寻求把流量推进到它的邻点中距顶点 t 最近的顶点。预流推进算法中用到一个高度函数 h 来确定推流边。

对于给定网络 $G=(V,E)$ 的一个流,其高度函数 h 是定义在 G 的顶点集 V 上的一个非负函数。该函数满足:

(1) 对于 G 的残流网络中的每一条边 (u,v),有 $h(u) \leqslant h(v)+1$。

(2) $h(t)=0$。

G 的残流网络中满足 $h(u)=h(v)+1$ 的边 (u,v) 称为 G 的可推流边。

下面的函数 heights() 以每个顶点在残流网络中到汇点 t 的最短路长作为其高度函数值来构造一个有效的高度函数。

```
void heights()
{
    QUEUE<int> queue(G.V());
    for (int i = 0; i < G.V(); i++) h[i]=0;
    queue.put(t);
    // 广度优先搜索
    while (! queue.empty())
    {
```

```
            int v = queue.get();
            // 搜索与顶点 v 相连的边
            adjIterator< Edge > A(G, v);
            for (Edge* e = A.beg(); !A.end(); e = A.nxt())
              {
                int w = e->other(v);
                if ( h[w] == 0 && e->residual(w))
                  {h[w] = h[v] + 1; queue.put(w); }
              }
          }
        h[s] = G.V();
      }
```

下面给出一般的预流推进算法的基本框架:一般的预流推进算法。

步骤 0 构造初始预流 flow。对源顶点 s 的每条出边 (s,v),令 $flow(s,v)=cap(s,v)$;对其余边 (u,v),令 $flow(u,v)=0$。构造一个有效的高度函数 h。

步骤 1 如果残流网络中不存在活顶点,则计算结束,已经得到最大流;否则,转步骤 2。

步骤 2 在网络中选取活顶点 v。如果存在顶点 v 的出边为可推流边,则选取一条这样的可推流边,并沿此边推流。否则,令 $h(v)=\min\{h(w)+1 \mid (v,w)$是当前残流网络中的边$\}$,并转步骤 1。

一般的预流推进算法的每次迭代是一次推进运算或一次高度重新标号运算。对于推进运算,如果推进的流量等于推流边上的残留容量,则称为饱和推进;否则称为非饱和推进。

算法终止时,网络中不含有活顶点。此时只有顶点 s 和 t 的存流量非零。所以,此时的预流实际上已经是一个可行流。又由于在算法预处理阶段已经令 $h(s)=n$,而高度函数在计算过程中不会减少,因此算法在计算过程中可以保证网络中不存在可增广路。根据增广路定理,算法终止时的可行流是一个最大流。

一般的预流推进算法并未给出如何选择活顶点和可推流边。不同的选择策略导致不同的预流推进算法。在基于顶点的预流推进算法中,选定一个活顶点后,算法沿该活顶点的所有推流边进行推流运算,直至无可推流边或该顶点的存流量变成 0 时为止。

2. 算法描述

基于顶点的预流推进算法可描述如下:

```
template <class Graph, class Edge> class MAXFLOW
{
    const Graph &G;
    int s, t;
    vector<int> h, wt, st;
    PQ<int> gQ;

    public:
```

```
    MAXFLOW(Graph &G, int s, int t,int &maxflow) :
    G(G), s(s), t(t), h(G.V(),0), wt(G.V(), 0), st(G.V(), 0),  gQ(G.V(),h)
    {
      heights();
      wtmax();
      gQ.put(s); st[s] = 1;
      while (! gQ.empty())
      {
        int v = gQ.get();
        st[v] = 0;
        discharge(v);
        relabel(v);
      }
      maxflow + = wt[t];
    }
};
```

算法中用一个广义队列 gQ 存储当前活顶点集合。向量 st 是活顶点标志,st[v] = 1 表示顶点 v 是活顶点。向量 wt 用于存储当前活顶点的存流量。

函数 heights()初始化高度函数。

函数 wtmax()对源顶点 s,计算 $wt[s] = \sum_{(s,v)\in E} cap(s,v)$。

```
void wtmax()
{
   adjIterator< Edge > A(G, s);
   for (Edge * e = A.beg(); ! A.end(); e = A.nxt())
     if (e- >from(s))wt[s] + = e- >cap();
}
```

函数 push(e,v,w,f)对可推流边(v,w)推进流量 f。

```
void push(Edge * e,int v,int w,int f)
{
   e- >addflowRto(w, f);
   wt[v] - = f; wt[w] + = f;
}
```

函数 discharge(v)对活顶点 v 执行基于顶点的预流推进运算。

```
void discharge(int v)
{
   adjIterator< Edge > A(G, v);
   for (Edge * e = A.beg(); ! A.end(); e = A.nxt()){
```

```
            int w = e->other(v);
            int cap = e->capRto(w);
            int P = cap < wt[v] ? cap : wt[v];
            if (P > 0 && (v == s || h[v] == h[w] + 1)){
              push(e,v,w,P);
              if ((w != s) && (w != t) && (!st[w]))
              { gQ.put(w); st[w] = 1; }
            }
        }
    }
```

函数 relabel(v)对活顶点 v 执行高度重新标号运算。

```
void relabel(int v)
{
    if (v != s && v != t && wt[v] > 0)
    {
        int hv = INT_MAX;
        adjIterator<Edge> A(G, v);
        for (Edge* e = A.beg(); !A.end(); e = A.nxt()){
          int w = e->other(v);
          if ( e->residual(v) && h[w] < hv) hv = h[w];
        }
        if ( hv < INT_MAX ) h[v] = hv + 1;
        gQ.put(v); st[v] = 1;
    }
}
```

3. 算法的计算复杂性

上面的基于顶点的预流推进算法用一个广义队列 gQ 存储当前活顶点集合。这个广义队列可以是通常的 FIFO 队列、LIFO 栈、随机化队列、随机化栈，或按各种优先级定义的优先队列。由此可见，上面的基于顶点的预流推进算法实际上包括了一大类算法。因此，算法的效率与广义优先队列的选择密切相关。

如果选用通常的 FIFO 队列，则在最坏情况下，预流推进算法求最大流所需的计算时间为 $O(mn^2)$，其中 m 和 n 分别为图 G 的边数和顶点数。

如果以顶点高度值为优先级，选用优先队列实现预流推进算法，则在最坏情况下，求最大流所需的计算时间为 $O(\sqrt{m}n^2)$。这个算法也称为最高顶点标号预流推进算法

近来已提出许多其他预流推进算法的实现策略，在最坏情况下，算法所需的计算时间已接近 $O(mn)$。

8.2.4 最大流问题的变换与应用

1. 一般网络的最大流问题

在一般情况下,网络可能有多个源和多个汇。此时,可将一般网络的最大流问题转换为与之等价的单源单汇网络的最大流问题。具体做法是:在原网络的基础上,增加一个虚源 s 和一个虚汇 t。如果原网络有 p 个源 $s_1, s_2, \cdots, s_p$ 和 q 个汇 $t_1, t_2, \cdots, t_q$,则在原网络中增加 p 条以 s 为起点的边 $(s, s_1), (s, s_2), \cdots, (s, s_p)$,以及 q 条以 t 为终点的边 $(t_1, t), (t_2, t), \cdots, (t_q, t)$。新增各边的容量分别定义为顶点 $s_1, s_2, \cdots, s_p$ 的流出量和顶点 $t_1, t_2, \cdots, t_q$ 的流入量。新网络的最大流对应于原网络的最大流。

2. 可行流问题

在有多个源点和多个汇点的网络中,给每个源点一个正的流量,给每个汇点一个负的流量,且所有源点和所有汇点流量的代数和为零。可行流问题要求判断对于给定的多源和多汇网络是否存在满足源点和汇点流量约束的可行流。可行流问题实际上是前面所说的一般网络的最大流问题,容易转换为如下标准的最大流问题。

```
template <class Graph, class Edge> class FEASIBLE
{
  const Graph &G;
  public:
    FEASIBLE(const Graph &G, vector<int> sd) : G(G)
    {
      int maxflow,ss,dd,supply,demand;
      Graph F(G.V()+2,1);
      for (int v = 0; v < G.V(); v++)
      {
        adjIterator<Edge> A(G, v);
        for (Edge* e = A.beg(); ! A.end(); e = A.nxt())
          F.insert(e);
      }
      int s = G.V(), t = G.V()+1;
      supply=0; demand=0;
      for (int i = 0; i < G.V(); i++)
      if (sd[i] >= 0){
        supply += sd[i];
        F.insert(new EDGE(s, i, sd[i]));
      }
      else{
        demand -= sd[i];
        F.insert(new EDGE(i, t, -sd[i]));
      }
      MAXFLOW<Graph, Edge>(F, s, t, maxflow);
```

```
        F.checksd(s,t,ss,dd);
        if(supply == ss) cout << "supply ok" << endl;
        else cout << "supply not enough" << endl;
        if(demand == dd) cout << "demand met" << endl;
        else cout << "demand not met" << endl;
        F.outflow();
    }
};
```

其中,函数 checksd(s,t,ss,dd)用于计算源点 s 的总流出量 ss 和汇点 t 的总流入量 tt。

```
void checksd(int s,int t, int &ss, int &dd)
{
    ss = 0; dd = 0;
    for (int i = 0;i < Vcnt;i++)
    {
      if (adj[s][i] && adj[s][i] -> from(s) && adj[s][i] -> flow() > 0 )
        ss += adj[s][i] -> flow();
      if (adj[i][t] && adj[i][t] -> from(i) && adj[i][t] -> flow() > 0 )
        dd += adj[i][t] -> flow();
    }
}
```

3. 网络的顶点容量约束

在有顶点容量约束的网络最大流问题中,除了需要满足边容量约束外,在网络的某些顶点处还要满足顶点容量约束,即流经该顶点的流量不能超过给定的约束值。这类问题很容易转换为标准的最大流问题。只要将有顶点容量约束的顶点 u 用一条边(u,v)来替换,原来顶点 u 的入边仍为顶点 u 的入边,原来顶点 u 的出边改为顶点 v 的出边。连接顶点 u 和顶点 v 只有一条边(u,v),其边容量为原顶点 u 的顶点容量。容易看出,变换后网络的最大流就是原网络中满足顶点约束的最大流。

4. 二分图的最大匹配问题

设 $G=(V,E)$是一个无向图。如果顶点集合 V 可分割为两个互不相交的子集 X 和 Y,并且图中每条边(i,j)所关联的两个顶点 i 和 j 分属于这两个不同的顶点集,则称图 G 为一个二分图。

图匹配问题可描述如下:设 $G=(V,E)$是一个图。如果 $M\subseteq E$,且 M 中任何两条边都不与同一个顶点相关联,则称 M 是 G 的一个匹配。G 的边数最多的匹配称为 G 的最大(基数)匹配。

二分图的最大匹配问题就是在已知图 G 是一个二分图的前提下,求 G 的最大匹配。

给定一个二分图 G 和将图 G 的顶点集合 V 分成互不相交的两部分的顶点子集 X 和 Y,如下构造与之相应的网络 F:

(1) 增加一个源 s 和一个汇 t。

(2) 从 s 向 X 的每一个顶点都增加一条边;从 Y 的每一个顶点都向 t 增加一条边。

(3) 原图 G 中的每一条边都改为相应的由 X 指向 Y 的有向边;

(4) 置所有边的容量为 1。

求网络 F 的最大流。在从 X 指向 Y 的边集中,流量为 1 的边对应于二分图中的匹配边。最大流值对应于二分图 G 的最大匹配边数。

具体算法可实现如下:

```
template <class Graph, class Edge> class BMATCHING
{
    const Graph &G;
    public:
      BMATCHING(const Graph &G, int N1) : G(G)
      {
        int s,t,maxflow;
        Graph F(G.V()+2,1);
        for (int v = 0; v < G.V(); v++)
        {
          adjIterator<Edge> A(G, v);
          for (Edge* e = A.beg(); !A.end(); e = A.nxt())
            F.insert(e);
        }
        s = G.V(); t = G.V()+1;
        for (int i = 0; i < N1; i++)
          F.insert(new EDGE(s, i, 1));
        for (i = N1; i < s; i++)
          F.insert(new EDGE(i, t, 1));
        MAXFLOW<Graph, Edge>(F, s, t, maxflow);
        for (i = 0; i < N1; i++)
        {
          adjIterator<Edge> A(F, i);
          for (EDGE* e = A.beg(); !A.end(); e = A.nxt())
          if (e->flow() == 1 && e->from(i) && e->capRto(i) == 1 )
          cout << e->v() << "->" << e->w() << endl;
        }
      }
};
```

上述算法中,原图为 G。顶点集 X 和 Y 分别为 $X=\{0,1,\cdots,N1-1\}$; $Y=\{N1,N1+1,\cdots,n-1\}$。

由于网络 F 的每条边的容量不超过 1,所以用增广路算法求其最大流所需的计算时间为 $O(mn)$,其中 m 和 n 分别为图 G 的边数和顶点数。从而用上述算法求二分图的最大匹配所需的计算时间为 $O(mn)$。

5. 带下界约束的最大流问题

前面讨论的网络最大流问题中每条边(v,w)都有一个容量约束$cap(v,w)$,它实际上是对边(v,w)上流量的一个上界约束。在更一般的情况下,除了边容量的上界约束外,还有边流量的下界约束,即对于每条边(v,w)还有一个边流量的下界约束$caplow(v,w)$。在这种情况下,对可行流 flow 的容量约束相应地改变为$caplow(v,w) \leqslant flow(v,w) \leqslant cap(v,w)$。表示这类网络的边结构进行如下相应改变:

```
class EDGE
{
    int pv, pw, pcap, pcaplow, pflow, pflag;
    public:
    EDGE(int v, int w, int caplow, int cap) :
        pv(v), pw(w), pcaplow(caplow), pcap(cap), pflow(0), pflag(0) {}
    int v() const { return pv; }
    int w() const { return pw; }
    int cap() const { return pcap; }
    int caplow() const { return pcaplow; }
    int flow() const { return pflow; }
    bool from (int v) const
    { return pv == v; }

    void sublow() { pcap -= pcaplow; }
    void addlow() { pcap += pcaplow; pflow += pcaplow; pflag = 1; }

    bool residual(int v) const
    { return (pv == v && pcap - pflow > 0 || pw == v && pflow > 0 ); }

    int other(int v) const
    { return from(v) ? pw : pv; }
    int capRto(int v) const
    { return from(v) ? pflow - pcaplow * pflag : pcap - pflow; }
    void addflowRto(int v, int d)
    { pflow += from(v) ? -d : d; }
};
```

对于带下界约束的最大流问题通常可分两个阶段求解。第1阶段先找满足约束条件的可行流,第2阶段将找到的可行流扩展成最大流。

第1阶段先将找满足约束条件的可行流问题转换成一个等价的循环可行流问题。变换方法是在原网络中增加一条容量充分大的边(t,s)。这条边将从s流到t的流量再送回到t构成一个循环流。原网络有可行流当且仅当新网络有循环可行流。

设 flow 是新网络的一个循环可行流,则

(1) 对每个 $v \in V$(包括 s, t),有

$$顶点\ v\ 的流出量 - 顶点\ v\ 的流入量 = 0$$

即
$$\sum_{(v,w)\in E} \text{flow}(v,w) - \sum_{(w,v)\in E} \text{flow}(w,v) = 0$$

(2) 对每一条边 $(v,w) \in E$, $\text{caplow}(v,w) \leqslant \text{flow}(v,w) \leqslant \text{cap}(v,w)$。

进一步对流进行变换,设对每一条边 $(v,w) \in E$, $x(v,w) = \text{flow}(v,w) - \text{caplow}(v,w)$,代入上述(1)和(2)得到如下结果:

(3) 对每个顶点 $v \in V$,有

$$\sum_{(v,w)\in E} x(v,w) - \sum_{(w,v)\in E} x(w,v) = \text{sd}(v)$$

$$\text{sd}(v) = \sum_{(w,v)\in E} \text{caplow}(w,v) - \sum_{(v,w)\in E} \text{caplow}(v,w)$$

(4) 对每一条边 $(v,w) \in E$, $x(v,w) \leqslant \text{cap}(v,w) - \text{caplow}(v,w)$。

容易看出, $\sum_{v\in V} \text{sd}(v) = 0$。因此,上述循环可行流问题实际上就是前面讨论过的一般网络中的可行流问题。

实现上述思想的算法 feasible(G, s, t, sd)描述如下:

```
void read(char * filename,int& s,int& t, vector<int>& sd)
{
    int i,j, n,m,caplow,cap;
    ifstream inFile;
    inFile.open(filename);
    inFile>> n >> m >> s >> t;
    for (int k=0;k<m;k++)
    {
        inFile>> i >> j >> caplow >> cap;
        sd[j]+=caplow; sd[i]-=caplow;
        insert(new EDGE(i, j, caplow, cap));
    }
    inFile.close();
}

void feasible(Graph &G, int s, int t, vector<int> sd)
{
    int ss,dd,supply,demand,maxflow=0;
    Graph F(G.V()+2,1);
    for (int v = 0; v < G.V(); v++)
    {
        adjIterator<Edge> A(G, v);
        for (Edge* e = A.beg(); !A.end(); e = A.nxt())
          if (e->from(v)) { e->sublow(); F.insert(e); }
    }
    F.insert(new EDGE(t, s, 0,INT_MAX));
```

```
    s = G.V(); t = G.V()+1;
    supply = 0; demand = 0;
    for (int i = 0; i < G.V(); i++)
      if (sd[i] >= 0){
        supply += sd[i];
        F.insert(new EDGE(s, i, 0,sd[i]));
      }
      else{
        demand -= sd[i];
        F.insert(new EDGE(i, t, 0, -sd[i]));
      }
      MAXFLOW<Graph, Edge>(F, s, t, maxflow);
      F.checksd(s,t,ss,dd);
      if(supply == ss) cout<<"supply ok"<<endl;
      else cout<<"supply not enough"<<endl;
      if(demand == dd) cout<<"demand met"<<endl;
      else cout<<"demand not met"<<endl;
      for (v = 0; v < G.V(); v++)
      {
         adjIterator<Edge> A(G, v);
         for (Edge* e = A.beg(); !A.end(); e = A.nxt())
           if (e->from(v)) e->addlow();
      }
  }
```

找到可行流 x 后，对每一条边$(v,w)\in E$，按照 $flow(v,w)=x(v,w)+caplow(v,w)$计算得到原网络的一个可行流 flow。在此可行流的基础上，进一步用增广路算法扩展为一个最大流。上述2阶段算法可描述如下：

```
template <class Graph, class Edge> class LOWER
{
  const Graph &G;
  public:
    LOWER(Graph &G, int s, int t, vector<int> sd) : G(G)
    {
      int maxflow = 0;
      feasible(G, s,t,sd);
      adjIterator<Edge> A(G, s);
      for (Edge* e = A.beg(); !A.end(); e = A.nxt())
        if (e->from(s)) maxflow += e->flow();
      MAXFLOW<Graph, Edge>(G, s, t, maxflow);
      cout<<endl<<"Maxflow ="<<maxflow<<endl<<endl;
      G.outflow();
    }
```

```
};
```

注意,在第 1 阶段中,残流网络中向后边(v, w)的容量是 flow(v, w);在第 2 阶段中,残流网络中向后边(v, w)的容量是 flow(v, w) - caplow(v, w),因此算法中用标志变量 pflag 表示算法的阶段。当算法在第 1 阶段时,pflag = 0;当算法在第 2 阶段时,pflag = 1。在网络边结构中函数 capRto(v)修改为

```
int capRto(int v) const
{ return from(v) ? pflow - pcaplow * pflag : pcap - pflow; }
```

6. 带下界约束的最小流问题

带下界约束的最小流问题是找网络中满足流量上下界约束的最小可行流。

与带下界约束的最大流算法类似,可用 2 阶段方法求解。第 1 阶段先找满足约束条件的可行流。第 2 阶段以 t 为源点,以 s 为汇点,用增广路算法反向求解可找到最小可行流。

带下界约束的最小流算法可描述如下:

```
template <class Graph, class Edge> class LOWER
{
  const Graph &G;
  public:
  LOWER(Graph &G, int s, int t, vector<int> sd) : G(G)
  {
    int maxflow = 0;
    feasible(G, s,t,sd);
    adjIterator<Edge> A(G, s);
    for (Edge* e = A.beg(); ! A.end(); e = A.nxt())
      if (e->from(s)) maxflow -= e->flow();
    MAXFLOW<Graph, Edge>(G, t, s, maxflow);
    cout<<endl<<"Maxflow = "<<(-maxflow)<<endl<<endl;
    G.outflow();
  }
};
```

7. 表格数据取整问题

给定一个 p 行 q 列的实数表格 $A=\{a_{ij}\}$,其第 i 行和第 j 列的和分别为 r_i 和 c_j。表格数据取整问题要求将所给的实数表格 A 变换为一个相应的整数表格 $B=\{b_{ij}\}$,使得

(1) $b_{ij}=\text{round}(a_{ij})$;

(2) $\sum_{j=1}^{q} b_{ij} = \text{round}(r_i)$, $\sum_{i=1}^{p} b_{ij} = \text{round}(c_j)$。

其中,round(x)是对实数 x 的取整运算,可以是下取整 floor(x),也可以是上取整 ceil(x)。

这个问题可以转换为一个带下界约束的可行流问题。对于给定的表格 A,构造网络 G 如

下。网络 G 中有 $p+q+2$ 个顶点 $\{s,t,v_1,v_2,\cdots,v_p;w_1,w_2,\cdots,w_q\}$ 和 $p*q+p+q$ 条边 $\{(v_i,w_j),(s,v_i),(w_j,t);i=1,2,\cdots,p;j=1,2,\cdots,q\}$。其中,边 (v_i,w_j) 的容量上下界分别为 floor(a_{ij}) 和 ceil(a_{ij});边 (s,v_i) 的容量上下界分别为 floor(r_i) 和 ceil(r_i);边 (w_j,t) 的容量上下界分别为 floor(c_j) 和 ceil(c_j)。

易知,网络 G 的一个可行流对应于表格数据取整问题的一个解。

8.3 最小费用流问题

8.3.1 最小费用流

1. 网络流的费用

在实际应用中,与网络流有关的问题不仅涉及流量,而且还有费用因素。此时网络的每一条边 (v,w) 除了给定容量 cap(v,w) 外,还定义了一个单位流量费用 cost(v,w)。对于网络中一个给定的流 flow,其费用定义为

$$\text{cost(flow)} = \sum_{(v,w)\in E} \text{cost}(v,w) \times \text{flow}(v,w)$$

对于给定网络 G 中的流,其费用可计算如下:

```
template <class Graph, class Edge>
static int cost(Graph &G)
{
    int x = 0;
    for (int v = 0; v < G.V(); v++)
    {
      adjIterator<Edge> A(G, v);
      for (Edge* e = A.beg(); !A.end(); e = A.nxt())
        if (e->from(v) && e->costRto(e->w()) < INT_MAX)
          x += e->flow() * e->costRto(e->w());
    }
    return x;
}
```

2. 最小费用流问题

对于一个给定的网络 G,要求 G 的一个最大费用流 flow,使流的总费用 $\text{cost(flow)} = \sum_{(v,w)\in E} \text{cost}(v,w) \times \text{flow}(v,w)$ 最小。

3. 最小费用可行流问题

对于一个给定的多源多汇网络 G,要求 G 的一个可行流 flow,使可行流的总费用 $\text{cost(flow)} = \sum_{(v,w)\in E} \text{cost}(v,w) \times \text{flow}(v,w)$ 最小。

前面已经讨论过,可行流问题等价于最大流问题。类似地,最小费用可行流问题也等价于

最小费用流问题。

8.3.2 消圈算法

1. 算法基本思想

在与最小费用流问题有关的算法中，仍然沿用残流网络的概念。此时，残流网络中边的费用定义为

```
int costRto(int v)
{
    return from(v) ? -pcost : pcost;
}
```

也就是说，当残流网络中的边是向前边时，其费用不变；当残流网络中的边是向后边时，其费用为原费用的负值。

由于残流网络中存在负费用边，所以残流网络中就不可避免地会产生负费用圈。而在与最小费用流问题有关的算法中，负费用圈是一个重要概念。

最小费用流问题的最优性条件定理 网络 G 的最大流 flow 是 G 的一个最小费用流的充分且必要条件是 flow 所相应的残流网络中没有负费用圈。

根据这一定理，可以设计出求最小费用流的消圈算法如下：

步骤 0 用最大流算法构造最大流 flow。

步骤 1 如果残流网络中不存在负费用圈，则计算结束，已经找到最小费用流；否则，转步骤 2。

步骤 2 沿找到的负费用圈增流，并转步骤 1。

2. 算法描述

求最小费用流的消圈实现算法如下：

```
template <class Graph, class Edge> class MINCOST
{
    Graph &G;
    int s, t;
    vector<int> wt;
    vector <Edge *> st
    public:
    MINCOST(Graph &G, int s, int t) : G(G), s(s), t(t), st(G.V()), wt(G.V())
    {
      int flow = 0;
      MAXFLOW<Graph, Edge>(G, s, t, flow);
      for (int x = negcyc(); x != -1; x = negcyc()) augment(x, x);
    }
};
```

算法中用向量 st 存储找到的负费用圈。算法的核心是找负费用圈算法 negcyc()：

```
int negcyc()
{
    for(int i = 0;i < G.V();i + + )
    { int neg = negcyc(i); if(neg > = 0) return neg; }
    return - 1;
}
```

其中，函数 negcyc(i)以顶点 i 为起点，用 Bellman-Ford 找负费用圈算法在残流网络中搜索负费用圈。

```
int negcyc(int ss)
{
  st.assign(G.V(),0); wt.assign(G.V(), INT_MAX);
  QUEUE<int> Q(2 * G.V()); int N = 0;
  wt[ss] = 0.0;
  Q.put(ss); Q.put(G.V());
  while (! Q.empty())
  {
    int v;
    while ((v = Q.get()) = = G.V())
    {
      if (N + + > G.V()) return - 1;
      Q.put(G.V());
    }
    adjIterator<Edge> A(G, v);
    for (Edge * e = A.beg(); ! A.end(); e = A.nxt())
    {
      int w = e - > other(v);
      if (e - > capRto(w) = = 0 ) continue;
      double P = wt[v] + e - > wt(w);
      if (P < wt[w])
      {
        wt[w] = P;
        // 开始搜索负费用圈
        for ( int node_test = v; (st[node_test]! = 0 && node_test! = ss);
          node_test = ST(node_test) )
          if ( ST(node_test) = = w ) { st[w] = e; return w; }
        st[w] = e; Q.put(w);
      }
    }
  }
  return - 1;
```

```
}
```

找到负费用圈后由 augment(x, x)从负费用圈的起点 x 开始,沿找到的负费用圈增流。

```
int ST(int v) const
{
    if (st[v] == 0) {cout << "error!" << endl; return 0;}
    else return st[v] -> other(v);
}

void augment(int s, int t)
{
    int d = st[t] -> capRto(t);
    for (int v = ST(t); v != s; v = ST(v))
      if (st[v] -> capRto(v) < d)
        d = st[v] -> capRto(v);
      st[t] -> addflowRto(t, d);
      for ( v = ST(t); v != s; v = ST(v))
        st[v] -> addflowRto(v, d);
}
```

3. 算法的计算复杂性

如果给定的网络中有 n 个顶点和 m 条边,且每条边的容量不超过 M,每条边的费用不超过 C。由于最大流的费用不超过 mCM,而每次消去负费用圈至少使得费用下降 1 个单位,因此最多执行 mCM 次找负费用圈和增流运算。用 Bellman-Ford 算法找 1 次负费用圈需要 $O(mn)$计算时间。因此,求最小费用流的消圈算法在最坏情况下需要 $O(m^2nCM)$计算时间。

8.3.3 最小费用路算法

1. 算法基本思想

上面的消圈算法首先找到网络中的一个最大流,然后通过消去负费用圈使费用降低。最小费用路算法不用先找最大流,而是用类似于求最大流的增广路算法的思想,不断在残流网络中寻找从源 s 到汇 t 的最小费用路,然后沿最小费用路增流,直至找到最小费用流。残流网络中从源 s 到汇 t 的最小费用路是残流网络中从 s 到 t 的以费用为权的最短路。

残流网络中边的费用定义为

$$\mathrm{wt}(v,w)=\begin{cases}\mathrm{cost}(v,w) & (v,w)\in P^{+}\\ -\mathrm{cost}(w,v) & (v,w)\in P^{-}\end{cases}$$

即当残流网络中边(v,w)是向前边时,其费用为$\mathrm{cost}(v,w)$;当(v,w)是向后边时,其费用为$-\mathrm{cost}(w,v)$。

按此思想,可以设计出求最小费用流的最小费用路算法如下:

步骤0 初始可行0流。

步骤1 如果不存在最小费用路，则计算结束，已经找到最小费用流；否则，用最短路算法在残流网络中找从 s 到 t 的最小费用可增广路，转步骤2。

步骤2 沿找到的最小费用可增广路增流，并转步骤1。

2. 算法描述

求最小费用流的最小费用路实现算法如下：

```
template <class Graph, class Edge> class MINCOST
{
    Graph &G;
    int s, t, flow;
    vector<int> wt;
    vector <Edge * > st;
    public:
    MINCOST(Graph &G, int s, int t, int se) :
      G(G), s(s), t(t), flow(se), st(G.V()), wt(G.V())
    {
      while(shortest()) augment(s, t);
    }
};
```

找最小费用路的算法由 shortest()实现如下：

```
bool shortest()
{
  st.assign(G.V(),0); wt.assign(G.V(), INT_MAX);
  QUEUE<int> Q(2 * G.V()); int N = 0;
  if(flow <= 0) return false;
  wt[s] = 0.0;
  Q.put(s); Q.put(G.V());
  while (! Q.empty())
  {
    int v;
    while ((v = Q.get()) == G.V())
    {
      if (N++ > G.V()) return (wt[t]<INT_MAX);
      Q.put(G.V());
    }
    adjIterator<Edge> A(G, v);
    for (Edge* e = A.beg(); ! A.end(); e = A.nxt())
    {
```

```
            int w = e->other(v);
            if (e->capRto(w) == 0 ) continue;
            int P = wt[v] + e->wt(w);
            if (P < wt[w]){ wt[w] = P; st[w] = e; Q.put(w); }
        }
    }
    return (wt[t] < INT_MAX);
}
```

找到最小费用路后由 augment(s, t)从 s 开始,沿找到的最小费用路增流。

```
int ST(int v) const
{
    if (st[v] == 0) {cout << "error!" << endl; return 0;}
    else return st[v] -> other(v);
}

void augment(int s, int t)
{
    int d = st[t] -> capRto(t);
    for (int v = ST(t); v != s; v = ST(v))
    if (st[v] -> capRto(v) < d) d = st[v] -> capRto(v);
    if (d > flow) d = flow;
    st[t] -> addflowRto(t, d);
    for ( v = ST(t); v != s; v = ST(v)) st[v] -> addflowRto(v, d);
    flow -= d;
}
```

3. 算法的计算复杂性

算法的主要计算量在于连续寻找最小费用路并增流。如果给定的网络中有 n 个顶点和 m 条边,且每条边的容量不超过 M,每条边的费用不超过 C。由于每次增流至少使得流值增加 1 个单位,因此最多执行 M 次找最小费用路算法。如果找 1 次最小费用路需要 $S(m,n,C)$ 计算时间,则求最小费用流的最小费用路算法需要 $O(MS(m,n,C))$ 计算时间。

8.3.4 网络单纯形算法

1. 算法基本思想

消圈算法的计算复杂度不仅与算法找到的负费用圈有关,而且与每次找负费用圈所需的时间有关。虽然网络单纯形算法是从解线性规划问题的单纯形算法演变而来,但从算法的运行机制来看,可以将网络单纯形算法看作另一类消圈算法。其基本思想是用一个可行支撑树结构来加速找负费用圈的过程。

对于给定的网络 G 和一个可行流,相应的可行支撑树定义为 G 的一棵包含所有弱流边

的支撑树。

网络单纯形算法的第一步就是构造可行支撑树。从一个可行流出发,不断找由弱流边组成的圈,然后沿找到的弱流圈增流,消除所有弱流圈。在剩下的所有弱流边中加入零流边或饱和边构成一棵可行支撑树。

在可行支撑树结构的基础上,网络单纯形算法通过顶点的势函数,巧妙地选择非树边,使它与可行支撑树中的边构成负费用圈。然后,沿找到的负费用圈增流。

定义了顶点的势函数 Φ 后,残流网络中各边 (v,w) 的势费用定义为

$$c*(v,w) = c(v,w) - (\Phi(v) - \Phi(w))$$

其中,$c(v,w)$ 是 (v,w) 在残流网络中的费用。

如果对可行支撑树中所有边 (v,w) 有 $c*(v,w) = 0$,则相应的势函数 Φ 是一个有效势函数。

对于一棵可行支撑树,如果将一条非树边加入可行支撑树,产生残流网络中的一个负费用圈,则称该非树边为一条可用边。

可用边定理　给定一棵可行支撑树及其上的一个有效势函数,非树边 e 是一条可用边的充分必要条件是,e 是一条有正势费用的饱和边,或 e 是一条有负势费用的零流边。

事实上,设 $e=(v,w)$。边 e 与树边 $t_1,t_2,\cdots,t_d$ 构成一个圈 cycle: $t_1,t_2,\cdots,t_d,t_1$。其中,$v=t_1,w=t_d$, − cycle: $t_1,t_d,\cdots,t_2,t_1$。按照边的势费用定义有

$$c(w,v) = c*(w,v) + \Phi(t_d) - \Phi(t_1)$$
$$c(t_1,t_2) = \Phi(t_1) - \Phi(t_2)$$
$$c(t_2,t_3) = \Phi(t_2) - \Phi(t_3)$$
$$\cdots\cdots$$
$$c(t_{d-1},t_d) = \Phi(t_{d-1}) - \Phi(t_d)$$

各式相加得

$$\text{cost(cycle)} = c*(w,v)$$

由此可见,e 是一条可用边当且仅当 cost(cycle) <0;当且仅当 $c*(w,v)<0$;当且仅当 e 是一条有正势费用的饱和边,或 e 是一条有负势费用的零流边。

最优性条件定理　对于给定网络 G 的可行流 flow 及相应的可行支撑树 T,如果不存在 T 的可用边,则 flow 是一个最小费用流。

事实上,如果不存在 T 的可用边,则由可用边的定义知残流网络中没有负费用圈。又由最小费用流问题的最优性条件知 flow 是一个最小费用流。

根据这一最优性条件定理,可以设计求最小费用流的网络单纯形算法如下:

步骤 0　构造 flow 为初始可行 0 流。构造相应的可行支撑树 T 和有效的顶点势函数。

步骤 1　如果不存在 T 的可用边,则计算结束,已经找到最小费用流;否则,转步骤 2。

步骤 2　选取 T 的一条可用边与 T 的树边构成负费用圈,沿找到的负费用圈增流,从 T 中删去一条饱和边或零流边,重构可行支撑树,并转步骤 1。

2. 算法描述

实现网络单纯形算法首先面临的是如何表示可行支撑树。可行支撑树需要支持如下 3 个

基本运算：

(1) 计算顶点势函数。

(2) 沿负费用圈增流。

(3) 删除一条树边或插入一条树边。

有多种数据结构可满足上述要求。较简单的一种数据结构是父指针向量。用父指针向量 st 存储支撑树中各边。向量单元 st[v] 中存储的边是支撑树中的一条指向根结点方向的边 (v, w)。结点 v 的父结点是 w；边 st[v] 的父边是 st[w]。

用此数据结构可实现如下求最小费用流的网络单纯形算法：

```
template <class Graph, class Edge> class MINCOST
{
    const Graph &G;
    int s, t, valid;
    vector<Edge * > st;
    vector<int> mark, phi;
    public:
      MINCOST(Graph &G, int s, int t, int se) :
      G(G), s(s), t(t), st(G.V()), mark(G.V(), -1), phi(G.V())
      {
        int m,c;
        upbound(m,c);
        m = m * G.V(); c = c * G.V();
        Edge *z = new EDGE(s, t, se, c);
        G.insert(z);
        z->addflowRto(t, se);
        dfsR(z,t);
        for (valid = 1; ; valid++)
        {
          phi[t] = z->costRto(s);
          mark[t] = valid;
          for (int v = 0; v < G.V(); v++)
            if (v != t) phi[v] = phiR(v);
          Edge *x = besteligible();
          int rcost = costR(x, x->v());
          if (full(x) && rcost <= 0  ||  empty(x) && rcost >= 0) break;
          update(augment(x), x);
        }
        G.remove(z); delete z;
      }
};
```

算法中的向量 phi 用于存储顶点势函数 Φ 的值。向量 mark 是计算势函数时用到的标记向量。参数 se 是流入源 s 的流量。算法开始计算前先在网络中增加一条虚边 $z=(s,t)$。该

边的容量为 se,费用充分大。初始时,该边中的流量为 se。这条边的作用是在网络最大流的流量小于 se 时,将多余的流通过该边送到汇 t。其中用下面的函数 upbound(m , c)来计算网络中的最大边容量和最大边费用:

```
void upbound(int &cap, int &cost)
{
    cap = 0;cost = 0;
    for (int i = 0;i < G.V();i + + )
      for (int j = 0;j < G.V();j + + ){
        if (G.edge(v, w)  & & cap < G.edge(v, w) - > cap())
          cap = G.edge(v, w) - > cap();
        if (G.edge(v, w)  & & cost < G.edge(v, w) - > cost())
          cost = G.edge(v, w) - > cost();

}
```

函数 dfsR(e , w)以边 e 的顶点 w 为根结点,用深度优先搜索算法建立初始支撑树。

```
void dfsR(Edge * e,int w)
{
  int v = e - > other(w);
  st[v] = e;
  mark[v] = 1; mark[w] = 1;
  dfs(v); dfs(w);
  mark.assign(G.V(), - 1);
}

// 从顶点 v 开始进行深度优先搜索
  void dfs(int v)
  {
    adjIterator < Edge >  A(G, v);
    for (Edge * e = A.beg(); ! A.end(); e = A.nxt()){
      int w = e - > other(v);
      if (mark[w] = =  - 1) { st[w] = e; mark[w] = 1; dfs(w); }
    }
}
```

按照有效顶点势函数的定义,对支撑树的所有边(v , w),有 $c*(v, w) = 0$,因此有 $\Phi(v) = \Phi(w) - c(v, w)$。函数 phiR(v)依此公式在支撑树中递归地计算顶点势函数的值 $\Phi(v)$。

在 e = st[v] = (v , w)时,函数 ST[v]返回 w。

```
int ST(int v) const
{
    if (st[v] = = 0) return 0;
```

```
        else return st[v] - > other(v);
    }

    int phiR(int v)
    {
        if (mark[v] = = valid) return phi[v];
        phi[v] = phiR(ST(v)) - st[v] - > costRto(v);
        mark[v] = valid;
        return phi[v];
    }
```

确定网络单纯形算法效率的一个重要因素是选取可用边的算法效率。有多种不同策略实现这个选择。下面的算法 besteligible()选取使负费用圈的费用绝对值最大的可用边。

函数 costR(e, v)计算边 e 的势费用。

```
    int costR(Edge * e, int v)
    {
        int R = e- > cost() + phi[e- > w()] - phi[e- > v()];
        return  e- > from(v) ? R : - R;
    }

    Edge * besteligible()
    {
        Edge * x = 0;
        for (int v = 0, min = INT_MAX; v < G.V(); v+ + )
        {
          adjIterator < Edge >  A(G, v);
          for (Edge * e = A.beg(); ! A.end(); e = A.nxt())
            if (e- > capRto(e- > other(v)) > 0)
              if (e- > capRto(v) = = 0)
                if (costR(e, v) < min)
                { x = e; min = costR(e, v); }
        }
        return x;
    }
```

选出可用边 x 后,由 augment(x)沿边 x 和支撑树的树边构成的负费用圈增流。完成增流后,返回负费用圈中的一条饱和边或零流边。

算法首先根据可用边 x 是饱和边还是零流边来确定负费用圈的方向(v, w)。然后,计算顶点 v 和 w 的最近公共祖先 r。负费用圈由边(v, w),支撑树中从顶点 w 到 r 的路,以及支撑树中从 r 到顶点 v 的路所组成。沿此负费用圈计算出最大可增流量 d,然后再一次沿负费用圈对每一条边增流 d。由于可行支撑树中有饱和边和零流边,因此,如果算法找到的负费用圈中含有这种边,则出现退化情况,即算法中的最大可增流量 $d=0$。在这种情况下,并没有实

际增流。因此算法可能会陷入无限循环。幸运的是有一种简单的方法可以避免循环。如果在选取支撑树删除边时,总选取最靠近根结点的那条边,则可以保证算法不会无限循环。下面的算法 augment(x)正是按照这种策略选取删除边的。

```
Edge *augment(Edge *x)
{
    int v = full(x)? x->w():x->v();  // 负费用圈的方向
    int w = x->other(v);
    int r = lca(v, w);               // 顶点 v 和 w 的最近公共祖先
    int d = x->capRto(w);
    for (int u = w; u != r; u = ST(u))
      if (st[u]->capRto(ST(u)) < d)
        d = st[u]->capRto(ST(u));
    for ( u = v; u != r; u = ST(u))
      if (st[u]->capRto(u) < d)
        d = st[u]->capRto(u);
    x->addflowRto(w, d); Edge* e = x;
    for ( u = w; u != r; u = ST(u))
    {
      st[u]->addflowRto(ST(u), d);
      if (st[u]->capRto(ST(u)) == 0) e = st[u];
    }
    for ( u = v; u != r; u = ST(u))
    {
      st[u]->addflowRto(u, d);
      if (st[u]->capRto(u) == 0) e = st[u];
    }
    return e;
}
```

其中,如下函数 full(x)和 empty(x)用于判别边 x 是饱和边还是零流边。函数 lca(v, w)用于计算顶点 v 和 w 的最近公共祖先。

```
    bool full(Edge *x)
{
    return (x->capRto(x->w())==0);
}

bool empty(Edge *x)
{
    return (x->capRto(x->v())==0);
}

int lca(int v, int w)
```

```
{
  mark[v] = ++valid; mark[w] = valid;
  while (v != w)
  {
    if (v != t) v = ST(v);
    if (v != t && mark[v] == valid) return v;
    mark[v] = valid;
    if (w != t) w = ST(w);
    if (w != t && mark[w] == valid) return w;
    mark[w] = valid;
  }
  return v;
}
```

由 augment(x)完成沿负费用圈增流后,返回的边 e 是支撑树中的一条饱和边或零流边。算法进一步将边 x 加入支撑树,并从支撑树中删去边 e,重构新的可行支撑树。这个任务由算法 update(e,x)来实现。设 $x=(u,v)$;$e=(a,b)$;顶点 u 和 v 的最近公共祖先是 r,则边 e 在支撑树中从顶点 u 到 r 的路上,或在支撑树中从顶点 v 到 r 的路上。当边 e 在支撑树中从顶点 u 到 r 的路上时,删除边 e 后,支撑树中从顶点 u 到顶点 a 的路上各边方向应该反转。同样,当边 e 在支撑树中从顶点 v 到 r 的路上时,删除边 e 后,支撑树中从顶点 v 到顶点 a 的路上各边方向应该反转。

```
bool onpath(int a, int b, int c)
{
    for (int i = a; i != c; i = ST(i)) if (i == b) return true;
    return false;
}

void reverse(int u, int x)
{
    Edge *e = st[u];
    for (int i = ST(u); e->other(i) != x; i = e->other(i))
    { Edge *y = st[i]; st[i] = e; e = y; }
}

void update(Edge *w, Edge *y)
{
    if(w == y) return;
    int u = y->w(), v = y->v(), x = w->w();
    if (x == t || st[x] != w) x = w->v();
    int r = lca(u, v);
    if (onpath(u, x, r))
    { reverse(u, x); st[u] = y; return; }
    if (onpath(v, x, r))
```

```
        { reverse(v, x); st[v] = y; return; }
    }
```

在上面的算法中,函数 onpath(a , b , c)用于判断顶点 b 是否在顶点 a 到顶点 c 的路上。函数 reverse(u , x)用于反转从顶点 u 到顶点 x 的路上各边的方向。

3. 算法的计算复杂性

如果给定的网络中有 n 个顶点和 m 条边,且每条边的容量不超过 M ,每条边的费用不超过 C。由于最大流的费用不超过 mCM ,而每次消去负费用圈至少使得费用下降 1 个单位,因此最多执行 mCM 次找负费用圈和增流运算。用网络单纯形算法找 1 次负费用圈需要 $O(m)$ 计算时间。因此,求最小费用流的网络单纯形算法在最坏情况下需要 $O(m^2(CM))$ 计算时间。

8.3.5 最小费用流问题的变换与应用

1. 带下界约束的最小费用流问题

与带下界约束的最大流问题类似,带下界约束的最小费用流问题也分 2 阶段求解。第 1 阶段先找满足约束条件的可行流;第 2 阶段将找到的可行流扩展成最小费最大流。

```
template <class Graph, class Edge> class LOWER
{
    const Graph &G;
    public:
    LOWER(Graph &G, int s, int t, int se, int te, vector<int> sd) : G(G)
    {
      int maxflow = 0;
      feasible(G, s,t,sd);
      adjIterator<Edge> A(G, s);
      for (Edge* e = A.beg(); ! A.end(); e = A.nxt())
        if (e->from(s)) maxflow + = e->flow();
      MINCOST<GRAPH<EDGE>, EDGE>(G, s, t, se);
      G.outflow();
    }
};
```

与带下界约束的最大流问题不同之处是第 2 阶段调用的是最小费用流算法。

2. 带下界约束的最小费用最小流问题

与带下界约束的最小流问题类似,带下界约束的最小费用最小流问题也分 2 阶段求解。第 1 阶段先找满足约束条件的可行流;第 2 阶段以 t 为源点,以 s 为汇点,用最小费用流算法反向求解可找到最小费用最小可行流。

```
template <class Graph, class Edge> class LOWER
```

```
{
    const Graph &G;
    public:
    LOWER(Graph &G, int s, int t, int se,int te,vector<int> sd) : G(G)
    {
      int maxflow = 0;
      feasible(G, s,t,sd);
      adjIterator<Edge> A(G, s);
      for (Edge* e = A.beg(); ! A.end(); e = A.nxt())
      if (e->from(s)) maxflow -= e->flow();
    MINCOST<GRAPH<EDGE>, EDGE>(G, t, s, se);
    G.outflow();
  }
};
```

与带下界约束的最小流问题不同之处是第2阶段调用的是最小费用流算法。

3. 最小权二分匹配问题

给定一个带权二分图 H,找出 H 的一个最小权二分匹配。这个问题也称为指派问题。

设 H 的二分顶点集为 $V1$ 和 $V2$。构造与 H 相应的网络 G 如下:

增设源点 s 和汇点 t。源点 s 到 $V1$ 中每个顶点有一条边,每条边的容量为1,费用为0。$V2$ 中每个顶点到汇点 t 有一条边,每条边的容量为1,费用为0。H 中每条边相应于 G 中一条边,该边的容量为1,费用为该边在 H 中的权。

易知,G 的最小费用流相应于 H 的一个最小权二分匹配。

上述变换可用如下算法实现:

```
template <class Graph, class Edge> class ASSIGNMENT
{
    const Graph &G;
    public:
      ASSIGNMENT(const Graph &G, int N1) : G(G)
      {
        int s,t,sum = 0;
        Graph F(G.V()+2,1);
        for (int v = 0; v < G.V(); v++)
        {
          adjIterator<Edge> A(G, v);
          for (Edge* e = A.beg(); ! A.end(); e = A.nxt())
          F.insert(e);
        }
      s = G.V(); t = G.V()+1;
      for (int i = 0; i < N1; i++) F.insert(new EDGE(s, i, 1,0));
      for (i = N1; i < s; i++) F.insert(new EDGE(i, t, 1,0));
```

```
            MINCOST<Graph, Edge>(F, s, t, N1);
            for (i = 0; i < N1; i++)
            {
              adjIterator<Edge> A(F, i);
              for (EDGE* e = A.beg(); !A.end(); e = A.nxt())
                if (e->flow() == 1 && e->from(i) )
                cout << e->v() << "->" << e->w() << endl;
            }
          }
        }
    };
```

4. 特殊线性规划问题

考察下面的特殊线性规划问题：

$$\min cx$$
$$\text{s.t.}\quad \boldsymbol{A}x \geqslant b$$
$$x \geqslant 0$$

其中，约束系数矩阵 $\boldsymbol{A}$ 具有特殊形式，即 $\boldsymbol{A}$ 是一个 0-1 矩阵，且 $\boldsymbol{A}$ 的每一列中的 1 是连续排列的。例如，8.1 节中讨论的仓库租赁问题就是这类线性规划问题。下面用一个简单例子说明算法思想。

$$\min\ cx$$
$$\text{s.t.}\quad \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix} x \geqslant \begin{bmatrix} 5 \\ 12 \\ 10 \\ 6 \end{bmatrix}$$
$$x \geqslant 0$$

引入松驰变量将不等式约束转换为等式约束。

$$\min\ cx$$
$$\text{s.t.}\quad \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & -1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \\ \\ y \\ \end{bmatrix} = \begin{bmatrix} 5 \\ 12 \\ 10 \\ 6 \\ 0 \end{bmatrix}$$
$$x, y \geqslant 0$$

其中，第 5 行是故意加入的恒等式 $0x + 0y = 0$。

从最后一行开始，每行减去上一行得

$$\min\ cx$$

$$\text{s.t.}\quad \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ \\ \\ y \\ \\ \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ -2 \\ -4 \\ -6 \end{bmatrix}$$

$$x, y \geqslant 0$$

此时的系数矩阵中,每一列有一个 1 和一个 -1,正好对应网络中一条边的起点和终点。

另外,右端矩阵各行数值的代数和为 0。由此,可构造相应的网络如图 8-7 所示。

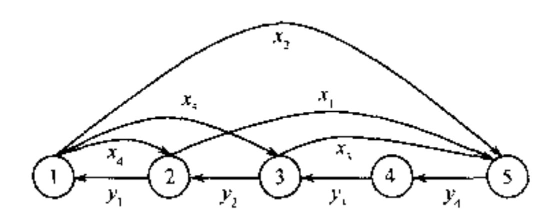


图 8-7　与特殊线性规划问题对应的网络

在一般情况下,特殊线性规划问题有 n 个变量和 m 个约束,则与特殊线性规划问题对应的网络中有 $m+1$ 个顶点和 $m+n$ 条边。设网络中的 $m+1$ 个顶点为 $1,2,\cdots,m+1$。约束矩阵 $\boldsymbol{A}$ 的每一行和每一列都对应于网络中一条边。例如,第 i 行对应于网络中的边 $(i+1,i)$,该边对应于松驰变量 y_i,其费用为 0。如果第 j 列中从第 p 行到第 q 行为连续的 1,其余各行为 0,则第 j 列对应于网络中的边 $(p,q+1)$,该边对应于变量 x_j,其费用为 c_j。网络中第 i 个顶点的流入(或流出)量为 $b_{i+1}-b_i$。

这是一个多源和多汇的网络,引入虚源 s 和虚汇 t 后,可将其变换为标准的单源单汇网络。该网络的一个最小费用流对应于特殊线性规划问题的一个解。

根据上面的讨论,可设计解特殊线性规划问题的算法如下:

```
class ConsecLP
{
    int ncon,   // 约束数
      nvar,   // 变量数
      * * a,s,t,supply;
    public:
    ConsecLP(char * filename)
    {
      read(filename);
      GRAPH<EDGE> G(ncon+3,1);
      constructG(G);
      MINCOST<GRAPH<EDGE>, EDGE>(G, s, t, supply);
      cout<<"Minicost = "<<cost<GRAPH<EDGE>, EDGE>(G)<<endl;
      G.outx();
    }
};
```

其中,read 读入特殊线性规划问题的参数;constructG 根据读入数据构造相应的网络 G,然后用最小费用流算法求解;函数 outx()根据最小费用流输出特殊线性规划问题的解。

构造网络的算法 constructG 描述如下:

```
void constructG(GRAPH < EDGE >  &G)
{
    int i,j,p,q,maxc = 0;
    a[ncon + 1][0] = 0;
    for (i = ncon + 1;i > 1;i - - )a[i][0] - = a[i - 1][0];
    for (i = 1;i < = ncon;i + + ) if (a[i][0] > maxc) maxc + = a[i][0];
    for (j = 1;j < = nvar;j + + )
    {
      p = 0;q = 0;
      for (i = 1;i < = ncon;i + + )
      {
        if ((p = = 0) & &(a[i][j] = = 1))p = i;
        if ((p > 0) & &(q = = 0) & &(a[i][j] = = 0))q = i;
      }
      if (q = = 0) q = ncon + 1;
      EDGE  * e = G.edge(p - 1, q - 1);
      if ( e = = 0 || e! = 0 & & e - > cost() > a[0][j] )
        G.insert(new EDGE(p - 1, q - 1, maxc,a[0][j],j));
    }
    for (i = 1;i < = ncon;i + + ) G.insert(new EDGE(i, i - 1, maxc,0,0));
    s = ncon + 1; t = ncon + 2; supply = 0;
    for (i = 1; i < = ncon + 1; i+ + )
      if (a[i][0] > = 0)
        {
          supply + = a[i][0];
          G.insert(new EDGE(s, i - 1, a[i][0],0,0));
        }
        else G.insert(new EDGE(i - 1, t, - a[i][0],0,0));
}
```

5. 最小逃脱问题

一个由 m 行 n 列的结点组成的栅格状无向图如图 8-8 所示。用(v,w)表示位于第 i 行第 j 列的结点。满足 $i=1$ 或 $i=m$ 或 $j=1$ 或 $j=n$ 的结点(v,w)是边界结点,其他结点为内部结点。在每一个内部结点处,都有 4 个其他结点与其相邻。对于栅格中 f 个给定的起始点 $(x_1,y_1),(x_2,y_2),\cdots,(x_f,y_f)$,逃脱问题要求确定是否存在从这 f 个起始点开始到栅格边界的 f 条不相交的路径。例如,图 8-8(a)的栅格有一个逃脱,而图 8-8(b)的栅格就没有逃脱。设每条栅格边的长度为 1。最小逃脱问题要求在所给栅格的所有逃脱中,找出逃脱路径总长度最短的一个逃脱。图 8-8(c)中的逃脱是一个最小逃脱。

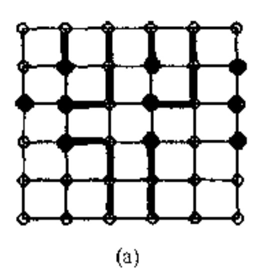
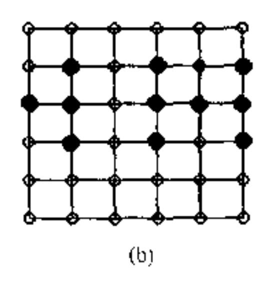
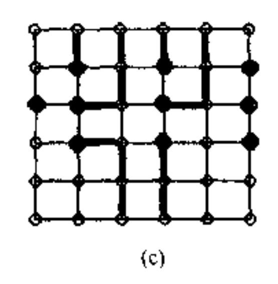

(a)

(b)

(c)

图 8-8 逃脱问题示例

对于给定的 $m \times n$ 栅格,构造相应的费用流网络 G 如下:

(1) 将每一个栅格点(v,w)拆成两个顶点 $v(i,j,1)$和 $v(i,j,2)$;用一条边$(v(i,j,1), v(i,j,2))$连接这两个顶点,并设该边的容量为 1,费用为 0,$1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$。

(2) 在一般情况下,将原来与栅格点(v,w)相邻的 4 个栅格点$(i-1,j)$,$(i+1,j)$,$(i,j-1)$和$(i,j+1)$变换为 8 个顶点,它们与顶点 $v(i,j,1)$和 $v(i,j,2)$的连接情况如图 8-9 所示。这 8 条边的容量和费用均为 1。当原栅格点(v,w)是起始栅格点时,没有进入顶点 $v(i,j,1)$的 4 条边。当原栅格点(v,w)是边界栅格点时,没有从顶点 $v(i,j,2)$发出的 4 条边。

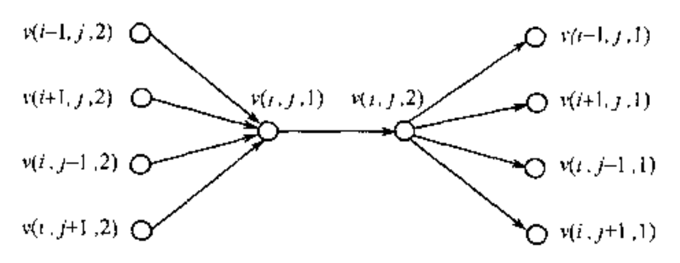


图 8-9 栅格点扩充变换

(3) 另外增设一个源 s 和一个汇 t。

(4) 对每一个起始栅格点(v,w)增加一条边$(s,v(i,j,1))$,其容量为 1,费用为 0。对每一个边界栅格点(v,w)增加一条边$(v(i,j,2),t)$,其容量为 1,费用为 0。

求网络 G 的最小费用最大流。当其流量为 f 时,求得的最小费用最大流即对应于一个最小逃脱,其最小费用即为所求的最小逃脱路径总长度。

如果仅要求判断是否有一个逃脱,只要求网络 G 的最大流。当流量为 f 时,所给栅格至少有一个逃脱。否则,所给栅格没有逃脱。

根据前面的分析,可以将原问题变换为一个网络最小费用流问题。用解网络最小费用流问题算法即可有效地找到给定栅格的最小逃脱。具体实现算法如下:

```
class ESCAPE
{
    int n,mm,nn,f,s,t, * st;
    int btype(int i,int j)
    {
      int b = 0;
      // b = 0 表示内部未占点;
      // b = 1 表示边界未占点;
      // b = 2 表示内部被占点;
      // b = 3 表示边界被占点;
```

```
  if ((i==1)||(i==mm)||(j==1)||(j==nn)) b++;
  if (start(v,w)) b+=2;
  return b;
}

int num(int i,int j)
{
  if ((i>=1)&&(i<=mm)&&(j>=1)&&(j<=nn)) return ((i-1)*nn+j)*2;
  else return -1;
}

int start(int i,int j)
{
  return st[(i-1)*nn+j];
}

void constructG(GRAPH<EDGE> &G)
{
  int u[5],v[5];
  for (int i=1;i<=mm;i++)
    for (int j=1;j<=nn;j++)
    {
      int k=num(v,w);
      int b=btype(v,w);
      u[1]=num(i-1,j); u[2]=num(i,j-1);
      u[3]=num(i+1,j); u[4]=num(i,j+1);
      v[1]=btype(i-1,j); v[2]=btype(i,j-1);
      v[3]=btype(i+1,j); v[4]=btype(i,j+1);
      G.insert(new EDGE(k, k+1,1,0));
      if (b>1) G.insert(new EDGE(s,k,1,0));
      else for (int x=1;x<5;x++)
            if ((u[x]>0)&&((v[x]==0)||(v[x]==2)))
              G.insert(new EDGE(u[x]+1,k,1,1));
      if ((b==1)||(b==3)) G.insert(new EDGE(k+1,t,1,0));
    }
}

void read(char *filename)
{
    int i,j;
    ifstream inFile;
    inFile.open(filename);
    inFile>> mm >> nn >> f;
    n=mm*nn*2+2;
```

```
        st = new int[mm * nn + 1];
        for (i = 0;i < = mm * nn;i + + ) st[i] = 0;
        s = 0; t = 1;
        for (int k = 0;k < f;k + + ){
            inFile > > i > > j;
            st[(i - 1) * nn + j] = 1;
        }
        inFile.close();
    }

    void trans(int i, int &u, int &v)
    {
        int k = i/2;
        u = k/nn;
        v = k%nn;
        if (v = = 0) v = nn;
        else u + + ;
    }

    void output(GRAPH < EDGE >  &G)
    {
        int u1,v1,u2,v2,sum = 0;
        adjIterator < EDGE >  A(G, s);
        for (EDGE * e = A.beg(); ! A.end(); e = A.nxt())
            if (e - > from(s) & &e - > flow() > 0) sum + + ;
        if (sum < f) {cout < < "No solution!" < < endl;return;}
        cout < < "succesful!" < < endl;
        sum = 0;
        for (int i = 2;i < n;i + + ){
            adjIterator < EDGE >  A(G, i);
            for (EDGE * e = A.beg(); ! A.end(); e = A.nxt())
                if (e - > from(i) & &e - > flow() > 0& &e - > cost() > 0){
                    trans(i,u1,v1);
                    trans(e - > w(),u2,v2);
                    cout < < "(" < < u1 < < "," < < v1 < < ") - - > (" < < u2 < < "," < < v2 < < ")" < < endl;
                    sum + + ;
                }
        }
        cout < < "Mincost = " < < sum < < endl;
    }
public:
    ESCAPE(char * filename)
    {
        read(filename);
```

```
        GRAPH<EDGE> G(n,1);
        constructG(G);
        MINCOST<GRAPH<EDGE>, EDGE>(G, s, t, f);
        output(G);
    }
};
```

习题8

8-1 试给出一个线性规划的例子,使其可行区域是无界的,但其最优目标函数值却是有限的。

8-2 试将单源最短路问题表示为一个线性规划问题。

8-3 试将网络最大流问题表示为一个线性规划问题。

8-4 试将网络最小费用流问题表示为一个线性规划问题。

8-5 运输计划问题。某集团公司拥有自己的产品运输网络。该公司现在生产 k 种不同的产品,每种产品都需要从其生产地运输到销售地。假设第 i 种产品的产地为 s_i,销售地为 t_i,需要的运输量为 d_i。集团公司需要规划其运输计划满足各种产品的运输需求。试建立该问题的线性规划模型。

8-6 试用单纯形算法解下面的线性规划问题:

$$\max\ z = x_1 + x_2 + x_3$$

$$\begin{aligned} \text{s.t.}\quad & 2x_1 + 7.5x_2 + 3x_3 \geqslant 10\ 000 \\ & 20x_1 + 5x_2 + 10x_3 \geqslant 30\ 000 \\ & x_1, x_2, x_3 \geqslant 0 \end{aligned}$$

8-7 边连通度问题。无向图 $G=(V,E)$的边连通度为 k 是指最少需要移去 G 的 k 条边才能使 G 成为不连通图。例如,树的边连通度为1;循环链的边连通度为2。试用网络最大流算法求给定图 G 的边连通度。

8-8 试证明有向无环网络的最大流问题等价于标准网络最大流问题。

8-9 试将无向网络最大流问题变换为标准网络最大流问题。

8-10 飞行员配对方案问题。第二次世界大战时期,英国皇家空军从沦陷国征募了大量外籍飞行员。由皇家空军派出的每一架飞机都需要配备在航行技能和语言上能互相配合的两名飞行员,其中1名是英国飞行员,另1名是外籍飞行员。在众多的飞行员中,每一名外籍飞行员都可以与其他若干名英国飞行员很好地配合。如何选择配对飞行的飞行员才能使一次派出最多的飞机。对于给定的外籍飞行员与英国飞行员的配合情况,试设计一个算法找出最佳飞行员配对方案,使皇家空军一次能派出最多的飞机。

8-11 太空飞行计划问题。W 教授正在为国家航天中心计划一系列的太空飞行。每次太空飞行可进行一系列商业性实验而获取利润。现已确定了一个可供选择的实验集合 $E=\{E_1,E_2,\cdots,E_m\}$和进行这些实验需要使用的全部仪器的集合 $I=\{I_1,I_2,\cdots,I_n\}$。实验 E_j需要用到的仪器是 I 的子集 $R_j\subseteq I$。配置仪器 I_k的费用为 c_k美元。实验 E_j的赞助商已同意

为该实验支付 p_j 美元。W 教授的任务是找出一个有效算法，确定在一次太空飞行中要进行哪些实验并因此而配置哪些仪器才能使太空飞行的净收益最大。这里净收益是指进行实验所获得的全部收入与配置仪器的全部费用的差额。

8-12 设 $G=(V,E)$ 是源为 s，汇为 t，且容量均为整数的一个流网络。已知 f 是 G 的一个最大流。

(1) 假设一条边 $(u,v)\in E$ 的容量增 1，试设计一个在 $O(|V|+|E|)$ 时间内更新最大流 f 的算法。

(2) 假设一条边 $(u,v)\in E$ 的容量减 1，试设计一个在 $O(|V|+|E|)$ 时间内更新最大流 f 的算法。

8-13 最小路径覆盖问题。给定有向图 $G=(V,E)$。设 P 是 G 的一个简单路(顶点不相交)的集合。如果 V 中每个顶点恰好在 P 的一条路上，则称 P 是 G 的一个路径覆盖。P 中路径可以从 V 的任何一个顶点开始，长度也是任意的，特别地，可以为 0。G 的最小路径覆盖是 G 的所含路径条数最少的路径覆盖。

设计一个有效算法求一个有向无环图 G 的最小路径覆盖。

[提示：设 $V=\{1,2,\cdots,n\}$，如下构造网络 $G1=(V1,E1)$

$$V1=\{x_0,x_1,\cdots,x_n\}\cup\{y_0,y_1,\cdots,y_n\}$$

$$E1=\{(x_0,x_i)\mid i\in V\}\cup\{(y_i,y_0)\mid i\in V\}\cup\{(x_i,y_j)\mid(i,j)\in E\}$$

求网络 $G1$ 的最大流。]

8-14 魔术球问题。假设有 n 根柱子，现要按下述规则在这 n 根柱子中依次放入编号为 1,2,3,…的球。

(1) 每次只能在某根柱子的最上面放球。

(2) 在同一根柱子中，任何两个相邻球的编号之和为完全平方数。

试设计一个算法，计算出在 n 根柱子上最多能放多少个球。例如，在 4 根柱子上最多可放 11 个球。

8-15 圆桌问题。假设有来自 n 个不同单位的代表参加一次国际会议。每个单位的代表数分别为 $r_i, i=1,2,\cdots,n$。会议餐厅共有 m 张餐桌，每张餐桌可容纳 $c_i(i=1,2,\cdots,m)$ 个代表就餐。为了使代表们充分交流，希望从同一个单位来的代表不在同一个餐桌就餐。试设计一个算法，给出满足要求的代表就餐方案。

8-16 混合图欧拉回路问题。试设计一个找混合图(既有无向边也有有向边的图)的欧拉回路的有效算法。

8-17 最长递增子序列问题。给定正整数序列 $x_0,x_1,\cdots,x_n$，要求

(1) 计算其最长递增子序列的长度 s。

(2) 设计一个有效算法，计算从给定的序列中最多可取出多少个长度为 s 的递增子序列。

8-18 试题库问题。假设一个试题库中有 n 道试题，每道试题都标明了所属类别，同一道题可能有多个类别属性。现要从题库中抽取 100 道题组成试卷，并要求试卷包含 20 类不同类型的试题。试设计一个满足要求的组卷算法。

8-19 机器人路径规划问题。假设机器人可在一个树状路径上自由移动。给定起点 s 和终点 t，机器人要从 s 运动到 t。树状路径上有若干可移动的障碍物。由于路径狭窄，任何时刻在路径的任何位置不能同时容纳两个物体。每一步可以将障碍物或机器人移到相邻的空结点上。设计一个有效算法用最少移动次数使机器人从 s 运动到 t。

8-20　方格取数问题。在一个有 n 个方格的棋盘中，每个方格中有一个正整数。现要从方格中取数，使任意两个数所在方格没有公共边，且取出的数的总和最大。试设计一个满足要求的取数算法。

8-21　餐巾计划问题。一个餐厅在相继的 N 天里，每天需用的餐巾数不尽相同。假设第 i 天需要 r_i 块餐巾($i=1,2,\cdots,N$)。餐厅可以购买新的餐巾，每块餐巾的费用为 p 分；或者把旧餐巾送到快洗部，洗一块需 m 天，其费用为 f 分；或者送到慢洗部，洗一块需 n 天($n>m$)，其费用为 $s<f$ 分。每天结束时，餐厅必须决定将多少块脏的餐巾送到快洗部，多少块餐巾送到慢洗部，以及多少块保存起来延期送洗。但是每天洗好的餐巾和购买的新餐巾数之和要满足当天的需求量。试设计一个算法为餐厅合理地安排好 N 天中餐巾的使用计划，使总的花费最小。

8-22　试将单源最短路问题表示为一个最小费用流问题。

8-23　试用最小费用流算法解中国邮路问题。

8-24　航空路线问题。给定一张航空图，图中顶点代表城市，边代表两城市间的直通航线。现要求找出一条满足下述限制条件的且途经城市最多的旅行路线：

(1) 从最西端城市出发，单向从西向东途经若干城市到达最东端城市，然后再单向从东向西飞回起点(可途经若干城市)。

(2) 除起点城市外，任何城市只能访问 1 次。

对于给定的航空图，试设计一个算法找出一条满足要求的最佳航空路线。

8-25　软件补丁问题。T 公司发现其研制的一个软件中有 n 个错误，随即为该软件发放了一批共 m 个补丁程序。每一个补丁程序都有其特定的适用环境，某个补丁只有在软件中包含某些错误而同时又不包含另一些错误时才可以使用。一个补丁在排除某些错误的同时，往往会加入另一些错误。换句话说，对于每一个补丁 i，都有两个与之相应的错误集合 $B1[i]$ 和 $B2[i]$，使得仅当软件包含 $B1[i]$ 中的所有错误，而不包含 $B2[i]$ 中的任何错误时，才可以使用补丁 i。补丁 i 将修复软件中的某些错误 $F1[i]$，而同时加入另一些错误 $F2[i]$。另外，每个补丁都耗费一定的时间。

试设计一个算法，利用 T 公司提供的 m 个补丁程序将原软件修复成一个没有错误的软件，并使修复后的软件耗时最少。

8-26　星际转移问题。由于人类对自然资源的消耗，人们意识到大约在 2300 年之后，地球就不能再居住了，于是在月球上建立了新的绿地，以便在需要时移民。令人意想不到的是，2177 年冬由于未知的原因，地球环境发生了连锁崩溃，人类必须在最短的时间内迁往月球。现有 n 个太空站位于地球与月球之间，且有 m 艘公共交通太空船在其间来回穿梭。每个太空站可容纳无限多的人，而每艘太空船 i 只可容纳 $H[i]$ 个人。每艘太空船将周期性地停靠一系列的太空站，例如，(1,3,4)表示该太空船将周期性地停靠太空站 134134134…。每一艘太空船从一个太空站驶往任一太空站耗时均为 1。人们只能在太空船停靠太空站(或月球、地球)时上、下船。初始时所有人全在地球上，太空船全在初始站。试设计一个算法，找出让所有人尽快全部转移到月球上的运输方案。

8-27　孤岛营救问题。1944 年，特种兵麦克接到国防部的命令，要求立即赶赴太平洋上的一个孤岛，营救被敌军俘虏的大兵瑞恩。瑞恩被关押在一个迷宫里，迷宫地形复杂，但幸好麦克得到了迷宫的地形图。迷宫的外形是一个长方形，其南北方向被划分为 N 行，东西方向被划分为 M 列，于是整个迷宫被划分为 $N\times M$ 个单元。每一个单元的位置可用一个有序数

对(单元的行号,单元的列号)来表示。南北或东西方向相邻的 2 个单元之间可能互通,也可能有一扇锁着的门,或者是一堵不可逾越的墙。迷宫中有一些单元存放着钥匙,并且所有的门被分成 P 类,打开同一类门的钥匙相同,打开不同类门的钥匙不同。

大兵瑞恩被关押在迷宫的东南角,即(N, M)单元里,并已经昏迷。迷宫只有一个入口,在西北角。也就是说,麦克可以直接进入(1,1)单元。另外,麦克从一个单元移动到另一个相邻单元的时间为 1,拿取所在单元钥匙的时间及用钥匙开门的时间可忽略不计。

试设计一个算法,帮助麦克以最快的方式到达瑞恩所在单元,营救大兵瑞恩。

8-28　汽车加油行驶问题。给定一个 $N \times N$ 的交通方形网格,设其左上角为起点◎,坐标为(1,1),X 轴向右为正,Y 轴向下为正,每个方格边长为 1,如图 8-10 所示。一辆汽车从起点◎出发驶向右下角终点▲,其坐标为(N, N)。在若干个网格交叉点处,设置了油库,可供汽车在行驶途中加油。汽车在行驶过程中应遵守如下规则:

(1) 汽车只能沿网格边行驶,装满油后能行驶 K 条网格边。出发时汽车已装满油,在起点与终点处不设油库。

(2) 当汽车行驶经过一条网格边时,若其 X 轴坐标或 Y 轴坐标减小,则应付费用 B,否则免付费用。

(3) 汽车在行驶过程中遇油库,应加满油并付加油费用 A。

(4) 在需要时可在网格点处增设油库,并付增设油库费用 C(不含加油费用 A)。

(5) (1)~(4)中的各数 N, K, A, B, C 均为正整数,且满足约束:$2 \leqslant N \leqslant 100, 2 \leqslant K \leqslant 10$。

设计一个算法,求出汽车从起点出发到达终点的一条所付费用最少的行驶路线。

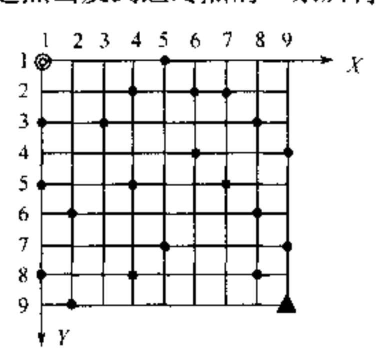


图 8-10　交通方形网格示例

第9章　NP完全性理论与近似算法

学习要点

· 理解RAM,RASP和图灵机计算模型
· 理解非确定性图灵机的概念
· 理解P类与NP类语言的概念
· 理解NP完全问题的概念
· 理解近似算法的性能比及多项式时间近似格式的概念
· 通过下面的范例学习NP完全问题的近似算法:
(1) 顶点覆盖问题
(2) 旅行售货员问题
(3) 集合覆盖问题
(4) 子集和问题

在计算机算法理论中,最深刻的问题之一是:从计算的观点来看,我们要解决的问题的内在复杂性如何,它是"易"计算的还是"难"计算的。如果我们知道了一个问题的计算时间下界,我们就可以较正确地评价解决该问题的各种算法的效率,进而确定对已有算法还有多少改进的余地。在许多情况下,要确定一个问题的内在计算复杂性是很困难的。已创造出的各种分析问题计算复杂性的方法和工具,可以较准确地确定许多问题的计算复杂性.

问题的计算复杂性可以通过解决该问题所需计算量的多少来度量。如何区分一个问题是"易"还是"难"呢? 人们通常将可在多项式时间内解决的问题看作是"易"解问题,而将需要指数函数时间解决的问题看作是"难"问题。这里所说的多项式时间和指数函数时间是针对问题的规模而言的,即解决问题所需的时间是问题规模的多项式还是指数函数。对于实际遇到的许多问题,人们至今无法确切地了解其内在的计算复杂性。因此只能用分类的方法将计算复杂性大致相同的问题归类进行研究. 而对于能够进行较彻底分析的问题则尽可能准确地确定其计算复杂性,从而获得对它的深刻理解。

9.1　计算模型

在进行问题的计算复杂性分析之前,首先必须建立求解问题所用的计算模型,包括定义该计算模型中所用的基本运算,其目的是为了使问题的计算复杂性分析有一个共同的客观尺度。

本节要讨论几个基本的计算模型。其中最重要的3个计算模型是随机存取机RAM(Random Access Machine)、随机存取存储程序机RASP(Random Access Stored Program Machine)以及图灵机(Turing Machine)。这3个计算模型在计算能力上是等价的,但计算速度不同。

9.1.1　随机存取机RAM

随机存取机RAM所描述的形式计算机是一台单累加器计算机。它不允许程序修改其自

身。RAM 由只读输入带、只写输出带、程序存储部件、内存储器和指令计数器 5 个部分组成。其中,内存储器中的 0 号寄存器用作累加器。

RAM 的结构如图 9-1 所示。

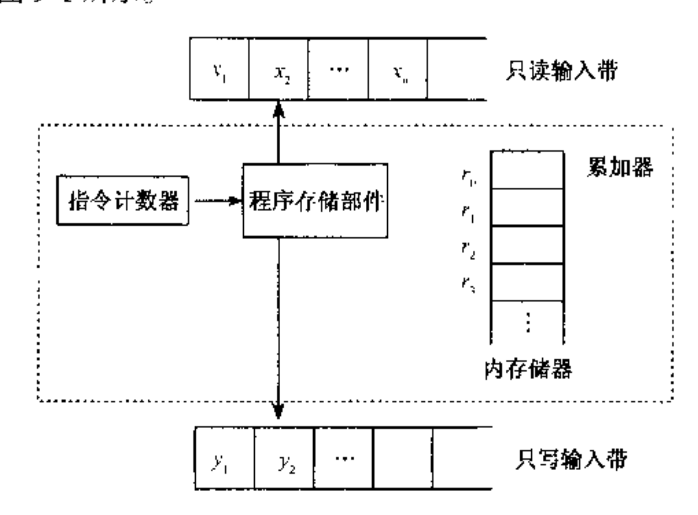


图 9-1　随机存取机 RAM

只读输入带由一系列方格组成,每格可存放一个整数(可为负)。从只读输入带读取一个数后,读写头向右移动一格。只写输出带的每个方格初始时为空。每执行一条写指令就在读写头下的方格中打印出一个整数,然后读写头右移一格。输出的符号一经写出,不能再修改。

内存储器由一系列寄存器 $r_0, r_1, \cdots, r_i, \cdots$ 组成。每个寄存器可以存放一个任意大小的整数。内存中寄存器的个数不受限制,也就是说,在程序中可以使用任意多个寄存器。这是 RAM 计算模型对现实计算机的一种抽象与简化。当所求解的问题规模不超过一台计算机的内存容量,且在计算中所出现的整数字长不超过计算机字长时,这种抽象是符合实际的。

RAM 程序不是存放在内存储器中,因而程序不能修改其自身。程序是一个带标号的指令序列。与现实计算机中所用的指令相仿,RAM 设有算术运算指令、输入输出指令、存取数指令及转移指令。RAM 中有直接寻址和间接寻址两种基本寻址方式。所有的计算在累加器 r_0 中进行。r_0 像其他寄存器一样,能容纳一个任意大小的整数。每条 RAM 指令由操作码和操作数两部分组成。其中操作数有以下 3 种形式:

(1) $=i$,操作数是整数 i 本身(直接数型)。

(2) i,i 是一非负整数,操作数是寄存器 r_i 的内容(直接地址型)。

(3) $*i$,i 为非负整数,若寄存器 r_i 的内容为整数 j,则操作数为寄存器 r_j 中的内容。当 j 为负整数时操作数无定义(间接地址型)。

RAM 对所有无定义的指令作停机处理。

设 c 是内存映射函数,即 $c(i)$ 表示寄存器 r_i 中的内容。以上 3 种类型的操作数的值 V 分别为:$V(=i)=i$, $V(i)=c(i)$, $V(*i)=c(c(i))$。

RAM 的基本指令集如表9-1 所示。

在 RAM 程序中,每执行过指令集中前 8 种指令之一后,指令计数器的值增 1。因此,RAM 程序中的指令是被顺序执行的,直至遇到指令集中后 4 种指令。JUMP 指令使程序无条件转移到标号所指指令处继续执行。JGTZ 指令在累加器中内容大于零时跳转,而 JZERO 指令在累加器中内容为零时跳转。

总的来讲,一个 RAM 程序定义了从输入带到输出带的一个映射。我们可以对这种映射

关系作不同的解释。常用的两个重要的解释是将这种映射关系看成是计算一个函数，或看成是接受一种语言。

表 9-1 RAM基本指令集

操作码	操作数	指令含义
(1) LOAD	$=i/i/*i$	取操作数入累加器
(2) STORE	$i/*i$	将累加器中数存入内存
(3) ADD	$=i/i/*i$	加法运算
(4) SUB	$=i/i/*i$	减法运算
(5) MULT	$=i/i/*i$	乘法运算
(6) DIV	$=i/i/*i$	除法运算
(7) READ	$i/*i$	读入
(8) WRITE	$=i/i/*i$	输出
(9) JUMP	标号	无条件转移到标号语句
(10) JGTZ	标号	正转移到标号语句
(11) JZERO	标号	零转移到标号语句
(12) HALT		停机

如果一个 RAM 程序 P 总是从输入带前 n 个方格中读入 n 个整数 $x_1,x_2,\cdots,x_n$，并且在输出带的第一个方格上输出一个整数 y 后停机，那么就说程序 P 计算了函数

$$f(x_1,x_2,\cdots,x_n)=y$$

对 RAM 程序的另一种解释是把它当作一个语言接受器。一个字母表是符号的有限集合，而语言是字母表上字符串的集合。字母表中的符号可以用整数 $1,2,\cdots,k$ 来表示。RAM 能以如下方式接受语言。

将字符串 $S=a_1a_2\cdots a_n$ 放在输入带上，在输入带的第 1 个方格中放入符号 a_1，第 2 个方格中放入符号 a_2，…，第 n 个方格中放入符号 a_n。然后在第 $n+1$ 个方格中放入 0，作为输入串的结束标志符。如果一个 RAM 程序 P 读了字符串 S 及结束标志符 0 后，在输出带的第 1 格输出一个 1 并停机，就说程序 P 接受了字符串 S。

P 可接受的语言 L 是 P 可接受的所有字符串的集合。对于不在 P 可接受的语言 L 中的输入串，程序 P 在输出带上输出一个不同于 1 的符号并停机，或者程序 P 永远不停机。

在 RAM 计算模型下，要精确地计算一个算法的时间和空间复杂性，就必须知道执行每条 RAM 指令所需的时间及每个寄存器实际所占的空间。在此，要讨论两种 RAM 程序的耗费标准：均匀耗费标准和对数耗费标准。

在均匀耗费标准下，每条 RAM 指令需要一个单位时间，每个寄存器占用一个单位空间。以后除特别注明外，RAM 程序的复杂性将按照均匀耗费标准来衡量。

对数耗费标准是基于这样的假定，即执行一条指令的耗费与以二进制表示的指令操作数长度成比例。在 RAM 计算模型下，假定一个寄存器可存放一个任意大小的整数。若设 $l(i)$ 是整数 i 所占的二进制位数，则

$$l(i)=\begin{cases}\lfloor\log|i|\rfloor & i\neq 0\\ 1 & i=0\end{cases}$$

各 RAM 指令的对数耗费如表9-2 所示。

表 9-2 RAM指令的对数耗费

RAM 指令	对数耗费
(1) LOAD a	$t(a)$
(2) STORE i	$l(c(0))+l(i)$

续表

RAM 指令	对数耗费
STORE $*i$	$l(c(0))+l(i)+l(c(i))$
(3) ADD a	$l(c(0))+t(a)$
(4) SUB a	$l(c(0))+t(a)$
(5) MULT a	$l(c(0))+t(a)$
(6) DIV a	$l(c(0))+t(a)$
(7) READ i	$l(input)+l(i)$
READ $*i$	$l(input)+l(i)+l(c(i))$
(8) WRITE a	$t(a)$
(9) JUMP b	1
(10) JGTZ b	$l(c(0))$
(11) JZERO b	$l(c(0))$
(12) HALT	1

其中,a 表示操作数,$t(a)$表示操作数 a 相应的耗费,b 表示标号。

对于前述 3 种类型的操作数,相应的对数耗费如表9-3 所示。

表 9-3 与 3 种类型操作数相应的对数耗费

操作数 a	对数耗费 $t(a)$
$=i$	$l(i)$
i	$l(i)+l(c(i))$
$*i$	$l(i)+l(c(i))+l(c(c(i)))$

9.1.2 随机存取存储程序机 RASP

由于 RAM 程序不是存储在 RAM 的存储器中,因而程序不能修改其自身。而随机存取存储程序机 RASP 计算模型,除了程序存储在存储器中并能修改其自身外,其他方面与 RAM 相似。

在 RASP 指令集中,由于不需要间接寻址,因而不允许使用间接地址。其余指令与 RAM 指令集一样。稍后我们会看到在程序执行过程中,RASP 可通过修改指令来模拟间接寻址。

RASP 的整体结构类似于 RAM,所不同的是 RASP 的程序是存储在寄存器中的。每条 RASP 指令占据两个连续的寄存器。第 1 个寄存器存放操作码的编码,第 2 个寄存器存放地址。RASP 指令用整数进行编码。表9-4 是 RASP 指令集中各指令的编码。

表 9-4 RASP 指令编码

指 令	编 码	指 令	编 码
LOAD i	1	DIV i	10
LOAD $=i$	2	DIV $=i$	11
STORE i	3	READ i	12
ADD i	4	WRITE i	13
ADD $=i$	5	WTITE $=i$	14
SUB i	6	JUMP i	15
SUB $=i$	7	JGTZ i	16
MULT i	8	JZERO i	17
MULT $=i$	9	HALT	18

开始时,RASP 程序装在存储器中,指令计数器设定在某个指定的寄存器上。寄存器中存储着操作码的编码。除转移指令外,每条指令执行后,指令计数器增加 2。当遇到 JUMP i(无条件转移),JGTZ i(当累加器中内容为正时转移)或 JZERO i(当累加器中内容为 0 时转移)指令时,指令计数器置为 i。这些指令的效果与相应的 RAM 指令相同。

与 RAM 程序类似，对于一个 RASP 程序，可在均匀耗费标准下或在对数耗费标准下来考虑其计算复杂性。在均匀耗费标准下，RASP 的计算复杂性与 RAM 相同。在对数耗费标准下，RASP 不仅要支付计算操作数的耗费，而且要支付存取指令本身的耗费。存取的耗费是 $l(LC)$。LC 是指令计数器中的内容。例如，执行存储在寄存器 j 和 $j+1$ 中的指令 ADD i 的对数耗费是 $l(j)+l(c(0))+l(i)$。执行存储在寄存器 j 和 $j+1$ 中的指令 ADD $=i$ 的对数耗费是 $l(j)+l(c(0))+l(i)+l(c(i))$。严格地讲，还应加上一项 $l(j+1)$，但这只差一个常数因子。今后我们只关心数量级，而对常数因子一般不予考虑。

解决同一问题的 RAM 程序和 RASP 程序有多大差别呢？实际上，不管是在均匀耗费标准下，还是在对数耗费标准下，RAM 程序和 RASP 程序的复杂性只差一个常数因子。在一个计算模型下，$T(n)$时间内完成的输入－输出映射可在另一个计算模型下模拟，并在 $kT(n)$时间内完成。其中，k 是一个常数因子。空间复杂性的情况也是类似的。

9.1.3 图灵机

图灵机是一个结构简单且计算能力很强的计算模型。

一台多带图灵机是由一个有限状态控制器和 k 条读写带($k \geqslant 1$)组成的。这些读写带的右端无限，每条带都从左到右划分为方格，每个方格可以存放一个带符号。带符号的总数是有限的。每条带上都有一个由有限状态控制器操纵的读写头或称为带头，它可以对这 k条带进行读写操作。有限状态控制器在某一时刻处于某种状态，且状态总数是有限的。图 9-2 是多带图灵机的示意图。

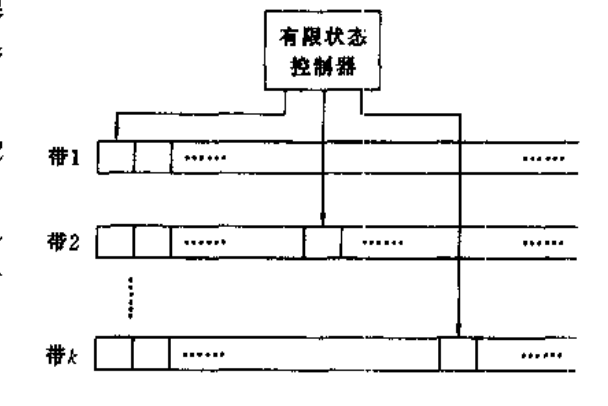


图 9-2　多带图灵机

根据有限状态控制器的当前状态及每个读写头读到的带符号，图灵机的一个计算步可实现下面 3 个操作之一或全部。

(1) 改变有限状态控制器中的状态。

(2) 清除当前读写头下的方格中原有带符号并写上新的带符号。

(3) 独立地将任何一个或所有读写头，向左移动一个方格(L)或向右移动一个方格(R)或停在当前单元不动(S)。

k 带图灵机可以形式化地描述为一个 7 元组(Q,T,I,δ,b,q_0,q_f)，其中：

(1) Q 是有限个状态的集合。

(2) T 是有限个带符号的集合。

(3) I 是输入符号的集合，$I\subseteq T$。

(4) b 是惟一的空白符，$b\in T-I$。

(5) q_0 是初始状态。

(6) q_f 是终止(或接受)状态。

(7) δ 是移动函数。它是从 $Q\times T^k$ 的某一子集映射到 $Q\times(T\times\{L,R,S\})^k$ 的函数。

对于某个包含一个状态及 k 个带符号的 $k+1$ 元组，移动函数将给出一个新的状态和 k 个序偶，每个序偶由一个新的带符号及读写头的移动方向组成。形式上可表述为

$$\delta(q,a_1,a_2,\cdots,a_k)=(q',(a'_1,d_1),(a'_2,d_2),\cdots,(a'_k,d_k))$$

当图灵机处于状态 q 且对一切 $1 \leq i \leq k$,第 i 条带的读写头扫描着的当前方格中的符号正好是 a_i 时,图灵机就按这个移动函数所规定的内容进行工作:

(1) 将图灵机的当前状态 q 改为状态 q'。

(2) 把第 i 条读写头下当前方格中的符号 a_i 清除并写上新的带符号 a'_i, $1 \leq i \leq k$。

(3) 按 d_i 指出的方向移动各带的读写头。这里 $d_i = \mathrm{L}$ 表示读写头左移一格, $d_i = \mathrm{R}$ 表示读写头右移一格, $d_i = \mathrm{S}$ 表示读写头不动。

一台图灵机可用来识别语言。一台图灵机的带符号集 T 应当包括这个语言的字母表中的全体符号和一个空白符 b,也许还有其他符号。开始时,第一条带上放有一个输入符号串,从最左的方格起每格放一个输入符号。这条带上其余方格都是空白。其他各带上也全是空白。所有读写头都处在各带左端的第一个方格上。当且仅当图灵机从指定的初始状态 q_0 开始,经过一系列计算步后,最终进入终止状态(或接受状态) q_f 时,称图灵机接受这个输入符号串。这台图灵机所能接受的所有输入符号串的集合,称作这台图灵机识别的一个语言。

与 RAM 模型类似,图灵机既可作为语言接受器,也可作为计算函数的装置。函数的自变量可编码成一字符串输入到一条输入带上,用一特殊符号 # 来隔开这些自变量。若图灵机经过有限步计算后,在一条指定的带上输出整数 y 并停机,则可以说图灵机计算出了 $f(x) = y$。由此可见,计算一个函数的过程与接受一个语言的过程没有什么区别。

图灵机 M 的时间复杂性 $T(n)$ 是它处理所有长度为 n 的输入所需的最大计算步数。如果对某个长度为 n 的输入,图灵机不停机, $T(n)$ 对这个 n 值无定义。图灵机的空间复杂性 $S(n)$ 是它处理所有长度为 n 的输入时,在 k 条带上所使用过的方格数的总和。如果某个读写头无限地向右移动而不停机, $S(n)$ 也无定义。

9.2 P 类与 NP 类问题

本书的许多算法都是多项式时间算法,即对规模为 n 的输入,算法在最坏情况下的计算时间为 $O(n^k)$, k 为一个常数。是否所有的问题都在多项式时间内可解呢? 回答是否定的。例如,存在一些不可解问题,如著名的“图灵停机问题”,任何计算机不论耗费多少时间也不能解该问题。此外,还有一些问题,虽然可以用计算机求解,但是对任意常数 k,它们都不能在 $O(n^k)$ 时间内得到解答。一般地,将可由多项式时间算法求解的问题看作是易处理的问题,而将需要超多项式时间才能求解的问题看作是难处理的问题。有许多问题,从表面上看似乎并不比排序或图的搜索等问题更困难,然而至今人们还没有找到解决这些问题的多项式时间算法,也没有人能够证明这些问题需要超多项式时间下界。也就是说,在图灵机计算模型下,这类问题的计算复杂性至今未知。为了研究这类问题的计算复杂性,人们提出了另一个能力更强的计算模型,即非确定性图灵机计算模型,简记为 NDTM (Nondeterministic Turing Machine)。在这个计算模型下,许多问题就可以在多项式时间内求解。

9.2.1 非确定性图灵机

在 9.1 节中介绍的图灵机计算模型中,移动函数 δ 是单值的,即对于 $Q \times T^k$ 中的每一个值,当它属于 δ 的定义域时, $Q \times (T \times \{L, R, S\})^k$ 中只有惟一的一个值与之对应。为了区别起见,称这种图灵机为确定性图灵机,简记为 DTM(Deterministic Turing Machine)。

一个 k 带的非确定性图灵机 M 也是一个 7 元组: $(Q, T, I, \delta, b, q_0, q_f)$。与确定性图灵

机不同的是，非确定性图灵机允许 δ 具有不确定性，即对于 $Q\times T^k$ 中的每一个值$(q,x_1,x_2,\cdots,x_k)$，当它属于 δ 的定义域时，$Q\times(T\times\{L,R,S\})^k$ 中有惟一的一个子集$\delta(q,x_1,x_2,\cdots,x_k)$与之对应。我们可以在 $\delta(q,x_1,x_2,\cdots,x_k)$中随意选定一个值作为它的函数值。这个不确定的函数 δ 仍称为移动函数。

k 带非确定性图灵机的瞬像与 k 带确定性图灵机的瞬像一样定义，它是一个 k 元组$(\alpha_1,\alpha_2,\cdots,\alpha_k)$。其中，$\alpha_i$ 是形如 xqy 的符号串。设非确定性图灵机 $M=(Q,T,I,\delta,b,q_0,q_f)$正处于状态 q，且第 i 个读写头$(1\leqslant i\leqslant k)$正扫描着第 i 条带上有符号 x_i 的方格。若有$(r,(y_1,D_1),\cdots,(y_k,D_k))\in\delta(q,x_1,x_2,\cdots,x_k)$，则说表达$(q,x_1,x_2,\cdots,x_k)$的瞬像（记为 B）与表达$(r,(y_1,D_1),\cdots,(y_k,D_k))$产生的瞬像（记为 C）之间有关系 $\vdash(M)$，记为 $B\vdash(M)C$（在不引起混淆时可略去(M)）。

如果对于每一个输入长度为 n 的可接受输入串，接受该输入串的非确定性图灵机 M 的计算路径长至多为 $T(n)$，则称 M 的时间复杂性是 $T(n)$。如果有某个导致接受状态的动作序列，在这个序列中，每一条带上至多扫描了 $S(n)$个不同的方格，则称 M 的空间复杂性为$S(n)$。

如前所述，确定性和非确定性图灵机的区别就在于，确定性图灵机的每一步只有一种选择，而非确定性图灵机却可以有多种选择。由此可见，非确定性图灵机的计算能力比确定性图灵机的计算能力强得多。对于一台时间复杂性为 $T(n)$的非确定性图灵机，可以用一台时间复杂性为 $O(C^{T(n)})$的确定性图灵机来模拟，其中 C 为一常数。这就是说，如果 $T(n)$是一个合理的时间复杂性函数，M 是一台时间复杂性为 $T(n)$的非确定性图灵机，可以找到一个常数 C 和一台确定性图灵机 M'，使得它们可接受的语言相同，且 M'的时间复杂性为$O(C^{T(n)})$。

9.2.2 P类与NP类语言

现在定义两个重要的语言类 P 和 NP 如下：

$P=\{L\mid L$ 是一个能在多项式时间内被一台 DTM 所接受的语言$\}$

$\mathrm{NP}=\{L\mid L$ 是一个能在多项式时间内被一台 NDTM 所接受的语言$\}$

由于一台确定性图灵机可看作是非确定性图灵机的特例，所以可在多项式时间内被确定性图灵机接受的语言也可在多项式时间内被非确定性图灵机接受，故 $P\subseteq\mathrm{NP}$。

虽然 P 和 NP 是借助图灵机来定义的，但也可以用其他计算模型来定义这两个语言类。直观上可以认为 P 是在多项式时间内的可识别的语言类。例如，在对数耗费标准下，如果图灵机接受语言 L 的时间复杂性为 $T(n)$，则 RAM 或 RASP 接受语言 L 的时间复杂性介于 $k_1T(n)$和 $k_2T^4(n)$之间，其中 k_1 和 k_2 都是正的常数。因此，$L\in P$ 当且仅当在 RAM 或 RASP 计算模型下存在接受语言 L 的多项式时间算法。

另一方面，若在 RAM 或 RASP 的指令系统上添加一条非确定性选择指令：

$$\mathrm{CHOICE}(L_1,L_2,\cdots,L_k)$$

也可以定义非确定性的 RAM 或 RASP 计算模型。CHOICE 指令非确定性地选出并执行标号为 $L_1,L_2,\cdots,L_k$ 的某个语句。因此，在对数耗费标准下，也可以用非确定性 RAM 或 RASP 模型来定义 NP 类。在该计算模型下，接受语言 L 的算法，称为非确定性算法。一个非确定性算法接受语言 L，当且仅当对每一个 $x\in L$，在该算法中存在一条接受 x 的计算路径。该算法的计算时间复杂性 $T(n)$就定义为，对所有长度为 n 的可接受输入串，其最短计算路径长度的最大值。因此，在非确定性 RAM 或 RASP 计算模型下，NP 类语言可定义为

NP = {L | L 是一个能在多项式时间内被一个非确定性 RAM 或 RASP 下算法所接受的语言}

下面考察 NP 类语言的一个例子,即无向图的团问题。该问题的输入是一个有 n 个顶点的无向图 $G=(V,E)$和一个整数 k。要求判定图 G 是否包含一个 k 顶点的完全子图(团),即判定是否存在 $V' \subseteq V$, $|V'|=k$,且对于所有的 $u,v \in V'$,有$(u,v) \in E$。

若用邻接矩阵来表示图 G,用二进制串表示整数 k,则团问题的一个实例可以用长度为 $n^2+\log k+1$ 的二进位串表示。因此,团问题可表示为语言

CLIQUE = {w # v | $w,v \in \{0,1\}^*$,以 w 为邻接矩阵的图 G 有一个 k 顶点的团,
v 是 k 的二进制表示}

接受该语言 CLIQUE 的非确定性算法如下:首先用非确定性选择指令选出包含 k 个顶点的候选顶点子集 V,然后确定性地检查该子集是否是团问题的一个解。算法分为 3 个阶段。

第 1 阶段将输入串 w # v 分解,并计算出 $n=\sqrt{|w|}$,以及用 v 表示的整数 k。若输入不具有形式 w # v 或 $|w|$ 不是一个平方数就拒绝该输入。显而易见,第 1 阶段可在 $O(n^2)$时间内完成。

在第 2 阶段中,非确定性地选择 V 的一个 k 元子集 $V' \subseteq V$。用一位向量 A[1:n]来表示该子集。A 中恰有 k 个 1,即 A[i] = 1 当且仅当 $i \in V'$。非确定性选择算法如下:

```
int j=0;
for (int i=1;i<=n;i++) {
  int m=Choice(0,1);
  switch(m)
    { case 0:A[i]=0;break;
      case 1:A[i]=1;j++;break;
    }
  }
if (j!=k) reject;
```

该算法产生 V 的一个 k 元子集 V',它的计算时间显然为 $O(n)$。因此,算法在第 2 阶段耗时 $O(n)$。

第 3 阶段是确定性地检查 V' 的团性质。若 V' 是一个团则接受输入,否则拒绝输入。这显然可以在 $O(n^4)$时间内完成。因此,整个算法的时间复杂性为 $O(n^4)$。

若图 $G=(V,E)$不包含一个 k 团,则在算法的第 2 阶段产生的任何 k 元子集 V' 不具有团性质。因此,算法没有导致接受状态的计算路径。反之,若图 G 含有一个 k 团 V',则算法的第 2 阶段中有一个计算路径产生 V',使得在算法的第 3 阶段导致接受状态。

综上即知,所述非确定性算法在多项式时间内接受语言 CLIQUE,即 CLIQUE $\in$ NP。

9.2.3 多项式时间验证

在识别语言 CLIQUE 的非确定性算法中,算法的第 2 阶段是非确定性的且耗时 $O(n)$。整个算法的计算时间复杂性主要取决于第 3 阶段的验证算法,即给定了图 G 的一个 k 团猜测 V',验证它是否确是一个团。若验证部分可在多项式时间内完成,则整个非确定性算法具有多项式时间复杂性,因而所识别的语言为 NP 类语言。这是识别 NP 类语言的非确定性算法所具有的一般特性。因此,我们也可以将 NP 类语言看作是在确定性计算模型下多项式时间

可验证的语言类。将验证算法定义为两个自变量的算法 A，其中一个自变量是通常的输入串 X，另一个自变量是一个称为"证书"的二进制串 Y。如果对任意串 $X\in L$，存在一个证书 Y，并且 A 可以用 Y 来证明 $X\in L$，则算法 A 就验证了语言 L。例如，在团问题中，证书是图 G 中一个 k 团，它提供了足够的信息供算法 A（第 3 阶段的算法）在多项式时间内验证语言 CLIQUE。因此，语言 CLIQUE 是多项式时间可验证语言。一般地，多项式时间可验证语言类 VP 可定义为

$VP=\{L\mid L\in\Sigma^*$，Σ 为一有限字符集，Σ^* 是 Σ 中字符构成的字符串的全体，存在一个多项式 p 和一个多项式时间验证算法 $A(X,Y)$，使得对任意 $X\in\Sigma^*$，$X\in L$ 当且仅当存在 $Y\in\Sigma^*$，$|Y|\leqslant p(|X|)$ 且 $A(X,Y)=1\}$

定理9-1 VP = NP。

证明：先证明 $VP\subseteq NP$。对于任意 $L\in VP$，设 p 是一个多项式，A 是一个多项式时间验证算法，则下面的非确定性算法接受语言 L：

(1) 对于输入 X，非确定性地产生一字符串 $Y\in\Sigma^*$；

(2) 当 $A(X,Y)=1$ 时接受 X。

该算法的步骤(1)与团问题的第 2 阶段的非确定性算法一样，至多在 $O(|X|)$ 时间内完成。步骤(2)的计算时间是 $|X|$ 和 $|Y|$ 的多项式，而 $|Y|\leqslant p(|X|)$。因此，它也是 $|X|$ 的多项式。整个算法可在多项式时间内完成。因此，$L\in NP$。由此可见 $VP\subseteq NP$。

反之，设 $L\in NP$，$L\in\Sigma^*$，且非确定性图灵机 M 在多项式时间 p 内接受语言 L。设 M 在任何情况下只有不超过 d 个的下一动作选择，则对于输入串 X，M 的任一动作序列可用 $\{0,1,\cdots,d-1\}$ 的长度不超过 $p(|X|)$ 的字符串来编码。不失一般性，设 $|\Sigma|\geqslant d$。验证算法 $A(X,Y)$ 用于验证"Y 是 M 上关于输入 X 的一条接受计算路径的编码"。即当 Y 是这样一个编码时，$A(X,Y)=1$。$A(X,Y)$ 显然可在多项式时间内确定性地进行验证，且

$$L=\{X\mid \text{存在 } Y \text{ 使得 } |Y|\leqslant p(|X|) \text{ 且 } A(X,Y)=1\}$$

因此 $L\in VP$。由此可知 $VP\supseteq NP$。

综上即知，VP = NP。

9.3 NP 完全问题

从 P 类和 NP 类语言的定义，我们已知道 $P\subseteq NP$。直观上看，P 类问题是确定性计算模型下的易解问题类，而 NP 类问题是非确定性计算模型下的易验证问题类。在通常情况下，解一个问题要比验证问题的一个解困难得多，特别在有时间限制的条件下更是如此。因此，大多数的计算机科学家认为 NP 类中包含了不属于 P 类的语言，即 $P\neq NP$。但这个问题至今没有获得明确的解答。也许使大多数计算机科学家相信 $P\neq NP$ 的最令人信服的理由是存在一类 NP 完全问题。这类问题有一种令人惊奇的性质，即如果一个 NP 完全问题能在多项式时间内得到解决，那么 NP 中的每一个问题都可以在多项式时间内求解，即 $P=NP$。尽管已进行了多年的研究，目前还没有一个 NP 完全问题有多项式时间算法

9.3.1 多项式时间变换

设 $L_1\subseteq\Sigma_1^*$，$L_2\subseteq\Sigma_2^*$ 是两个语言。所谓语言 L_1 能在多项式时间内变换为语言 L_2（简记为 $L_1\propto_p L_2$）是指存在映射 $f:\Sigma_1^*\rightarrow\Sigma_2^*$，且 f 满足：

(1) 有一个计算 f 的多项式时间确定性图灵机;

(2) 对于任意的 $x \in \Sigma_1^*$, $x \in L_1$ 当且仅当 $f(x) \in L_2$。

定义 语言 L 是 NP 完全的当且仅当

(1) $L \in$ NP;

(2) 对于所有 $L' \in$ NP 有 $L' \propto_p L$。

如果有一个语言 L 满足上述性质(2),但不一定满足性质(1),则称该语言是 NP 难的。所有 NP 完全语言构成的语言类称为 NP 完全语言类,记为 NPC。由 NPC 类语言的定义可以看出,它们是 NP 类中最难的问题,也是研究 P 类与 NP 类的关系的核心所在。

定理9-2 设 L 是 NP 完全的,则

(1) $L \in P$ 当且仅当 $P =$ NP;

(2) 若 $L \propto_p L_1$,且 $L_1 \in$ NP,则 L_1 是 NP 完全的。

证明:(1) 若 $P =$ NP,则显然 $L \in P$。反之,设 $L \in P$,而 $L_1 \in$ NP,则 L 可在多项式时间 p_1 内被确定性图灵机 M 所接受。又由 L 的 NP 完全性知 $L_1 \propto_p L$,即存在映射 f,使 $L = f(L_1)$。

设 N 是在多项式时间 p_2 内计算 f 的确定性图灵机。我们用图灵机 M 和 N 构造识别语言 L_1 的算法 A 如下:

① 对于输入 x,用 N 在 $p_2(|x|)$时间内计算出 $f(x)$;

② 在时间$|f(x)|$内将读写头移到 $f(x)$的第一个符号处;

③ 用 M 在时间$p(f|x|)$内判定 $f(x) \in L$。若 $f(x) \in L$,则接受 x,否则拒绝 x。

上述算法显然可接受语言 L_1,其计算时间为 $p_2(|x|) + |f(x)| + p_1(f|x|)$。由于图灵机一次只能在一个方格中写入一个符号,故$|f(x)| \leqslant |x| + p_2(|x|)$。因此,存在多项式 r 使得 $p_2(|x|) + |f(x)| + p_1(f|x|) \leqslant r(x)$。因此,$L_1 \in P$。由 L_1 的任意性即知 $P =$ NP。

(2) 只要证明对任意的 $L' \in$ NP,有 $L' \propto_p L_1$。由于 L 是 NP 完全的,故存在一个多项式时间变换 f 使 $L = f(L')$。又由于 $L \propto_p L_1$,故存在一多项式时间变换 g 使 $L_1 = g(L)$。因此,若取 f 和 g 的和复合函数 $h = g(f)$,则 $L_1 = h(L')$。易知 h 为一多项式。因此 $L' \propto_p L_1$。由 L' 的任意性即知 $L_1 \in$ NPC。

从定理9-2(1)可知,如果任一 NP 完全问题可在多项式时间内求解,则所有 NP 中的问题都可在多项式时间内求解。反之,若 $P \neq$ NP,则所有 NP 完全问题都不可能在多项式时间内求解。

定理9-2(2)实际上是证明问题的 NP 完全性的有力工具。一旦建立了问题 L 的 NP 完全性后,对于 $L_1 \in$ NP,只要证明问题 L 可在多项式时间内变换为 L_1,即 $L \propto_p L_1$,就可证明 L_1 也是 NP 完全的。

9.3.2 一些典型的 NP 完全问题

定理 9-2 所提供的证明问题的 NP 完全性的方法只有在有了第一个 NP 完全问题之后才能获得。获得"第一个 NP 完全问题"称号的是布尔表达式的可满足性问题,这就是著名的 Cook 定理:布尔表达式的可满足性问题 SAT 是 NP 完全的。Cook 定理的重要性是明显的,它给出了第一个 NP 完全问题。使得对于任何问题 Q,只要能证明 $Q \in$ NP 且 SAT $\propto_p Q$,就有 $Q \in$ NPC。所以,人们很快就证明了许多其他问题的 NP 完全性。这些 NP 完全问题都是直接或间接地以 SAT 的 NP 完全性为基础而得到证明的。由此逐渐生长出一棵以 SAT 为树根的

NP完全问题树。其中每个结点代表一个NP完全问题,该问题可在多项式时间内变换为它的任一儿子结点表示的问题。实际上,由树的连通性及多项式在复合变换下的封闭性可知,NP完全问题树中任一结点表示的问题可以在多项式时间内变换为它的任一后裔结点表示的问题。目前这棵NP完全问题树上已有几千个结点,并且还在继续生长。

下面介绍这棵NP完全树中的几个典型的NP完全问题。

(1) 合取范式的可满足性问题CNF-SAT

给定一个合取范式 α,判定它是否可满足

如果一个布尔表达式是一些因子和之积,则称之为合取范式,简称CNF(Conjunctive Normal Form)。这里的因子是变量 x 或 $\bar{x}$。例如 $(x_1+x_2)(x_2+x_3)(\bar{x}_1+\bar{x}_2+x_3)$ 就是一个合取范式,而 $x_1x_2+x_3$ 就不是合取范式。

(2) 三元合取范式的可满足性问题3-SAT

给定一个三元合取范式 α,判定它是否可满足。

(3) 团问题CLIQUE

给定一个无向图 $G=(V,E)$ 和一个正整数 k,判定图 G 是否包含一个 k 团,即是否存在 $V'\subseteq V$, $|V'|=k$,且对任意 $u,w\in V'$ 有 $(u,w)\in E$。

(4) 顶点覆盖问题VERTEX-COVER

给定一个无向图 $G=(V,E)$ 和一个正整数 k,判定是否存在 $V'\subseteq V$, $|V'|=k$,使得对于任意 $(u,v)\in E$ 有 $u\in V'$ 或 $v\in V'$。如果存在这样的 V',就称 V' 为图 G 的一个大小为 k 的顶点覆盖。

(5) 子集和问题SUBSET-SUM

给定整数集合 S 和一个整数 t,判定是否存在 S 的一个子集 $S'\subseteq S$,使得 S' 中整数的和为 t。

例如,若 $S=\{1,4,16,64,256,1040,1041,1093,1284,1344\}$ 且 $t=3754$,则子集 $S'=\{1,16,64,256,1040,1093,1284\}$ 是一个解。

(6) 哈密顿回路问题HAM-CYCLE

给定无向图 $G=(V,E)$,判定其是否含有一哈密顿回路。

(7) 旅行售货员问题TSP

给定一个无向完全图 $G=(V,E)$ 及定义在 $V\times V$ 上的一个费用函数 c 和一个整数 k,判定 G 是否存在经过 V 中各顶点恰好一次的回路,使得该回路的费用不超过 k。

9.4 NP完全问题的近似算法

迄今为止,所有的NP完全问题都还没有多项式时间算法。然而有许多NP完全问题具有很重要的实际意义,经常会遇到。对于这类问题,通常可采取以下几种解题策略。

(1) 只对问题的特殊实例求解。遇到一个NP完全问题时,应仔细考察是否必须在最一般的意义下求解。也许只要针对某种特殊情形求解就够了。而在特殊情形下常可得到高效算法。

(2) 用动态规划法或分支限界法求解。动态规划法和分支限界法是解许多NP完全问题的有效方法。在许多情况下,它们比穷举搜索法要有效得多。

(3) 用概率算法求解。有时可通过概率分析法来证明某个NP完全问题的"难"实例是很稀少的。因此可用概率算法来解这类NP完全问题,设计出在平均情况下的高效算法。

(4) 只求近似解。由于问题的输入数据通常是用测量的方法得到的，因此输入数据本身就是近似的。在实际中遇到的 NP 完全问题因此也不要求一定要获得非常精确的解答，只要求在一定的误差范围内的近似解就够了。许多解 NP 完全问题的近似算法可以用很少的时间获得一个很好的近似解。这是在实践中解决 NP 完全问题的非常有效且实用的方法。

(5) 用启发式方法求解。在用别的方法都不能奏效时，也可采用启发式算法来解 NP 完全问题。这类方法根据具体问题设计一些启发式搜索策略来寻求问题的解。在实际使用时可能很有效，但很难说清它的道理。

本章主要讨论解决 NP 完全问题的近似算法。

9.4.1 近似算法的性能

许多 NP 完全问题实质上是最优化问题，即要求使某个目标函数达到最大值或最小值的解。不失一般性，对于确定的问题，假设其每一个可行解所对应的目标函数值均不小于一个确定的正数。

若一个最优化问题的最优值为 c^*，求解该问题的一个近似算法求得的近似最优解相应的目标函数值为 c，则将该近似算法的性能比定义为

$$\eta = \max\left\{\frac{c}{c^*}, \frac{c^*}{c}\right\}$$

在通常情况下，该性能比是问题输入规模 n 的一个函数 $\rho(n)$，即

$$\max\left\{\frac{c}{c^*}, \frac{c^*}{c}\right\} \leqslant \rho(n)$$

这个定义对于极小化问题和极大化问题都是适用的。对于一个极大化问题，$0 < c \leqslant c^*$。此时近似算法的性能比，表示最优值 c^* 比近似最优值 c 大多少倍。对于一个极小化问题，$0 < c^* \leqslant c$。此时，近似算法的性能比表示近似最优值 c 比最优值 c^* 大多少倍。由 $c/c^* < 1$ 可推出 $c^*/c > 1$，故近似算法的性能比不会小于 1。一个能求得精确最优解的算法的性能比为 1。在通常情况下，近似算法的性能比大于 1。近似算法的性能比越大，它求出的近似最优解就越差。

有时用相对误差来表示一个近似算法的精确程度会更方便些。若最优化问题的精确最优值为 c^*，而一个近似算法求出的近似最优值为 c，则该近似算法的相对误差定义为

$$\lambda = \left|\frac{c - c^*}{c^*}\right|$$

近似算法的相对误差总是非负的。若对问题的输入规模 n，有一个函数 $\varepsilon(n)$ 使得

$$\left|\frac{c - c^*}{c^*}\right| \leqslant \varepsilon(n)$$

则称 $\varepsilon(n)$ 为该近似算法的相对误差界。近似算法的性能比 $\rho(n)$ 与相对误差界 $\varepsilon(n)$ 之间显然有关系：$\varepsilon(n) \leqslant \rho(n) - 1$。

有许多问题的近似算法具有固定的性能比或相对误差界，即 $\rho(n)$ 或 $\varepsilon(n)$ 是不随 n 的变化而变化的。此时，我们用 ρ 和 ε 来记性能比和相对误差界，表示它们不依赖于 n。当然，还有许多问题没有固定性能比的多项式时间近似算法，其性能比只能随着输入规模 n 的增长而增大。

对有些 NP 完全问题，可以找到这样的近似算法，其性能比可以通过增加计算量来改进。也就是说在计算量和解的精确度之间有一个折衷。较少的计算量得到较粗糙的近似解，而较多的计算量可以获得较精确的近似解。

一个最优化问题的近似格式是指带有近似精度 $\varepsilon > 0$ 的一类近似算法 对于固定的 $\varepsilon > 0$，该近似格式表示的近似算法的相对误差界为 ε。若对固定的 $\varepsilon > 0$ 和问题的一个输入规模为 n 的实例，用近似格式表示的近似算法是多项式时间算法，则称该近似格式为多项式时间近似格式。

多项式时间近似格式的计算时间不应随 ε 的减少而增长得太快。在理想情况下，若 ε 减少某一常数倍，近似格式的计算时间增长也不超过某一常数倍。换句话说，我们希望近似格式的计算时间是 $1/\varepsilon$ 和 n 的多项式。

当一个问题的近似格式的计算时间是关于 $1/\varepsilon$ 和问题实例的输入规模 n 的多项式时，称该近似格式为一完全多项式时间近似格式，其中，ε 是该近似格式的相对误差界。

下面针对一些常见的 NP 完全问题来研究有效近似算法的设计与分析方法。

9.4.2 顶点覆盖问题的近似算法

一个无向图 $G = (V, E)$ 的顶点覆盖是它的顶点集 V 的一个子集 $V' \subseteq V$，使得若 (u, v) 是 G 的一条边，则 $v \in V'$ 或 $u \in V'$。顶点覆盖 V' 的大小是它所包含的顶点个数 $|V'|$。

前面我们将顶点覆盖问题表述为一个判定问题，并证明了它的 NP 完全性。最优化形式的顶点覆盖问题是要找出图 G 的最小顶点覆盖。由于与其相应的判定问题是 NP 完全的，故最优化形式的顶点覆盖问题是 NP 难的。虽然要找到 G 的一个最小顶点覆盖可能是很困难的，但要找到一个近似最优的顶点覆盖却不太困难。下面的近似算法以无向图 G 为输入，并计算出 G 的近似最优顶点覆盖，可以保证计算出的近似最优顶点覆盖的大小不会超过最小顶点覆盖大小的 2 倍。

```
VertexSet approxVertexCover ( Graph g )
{
   cset = ∅;
   e1 = g.e;
   while (e1 ! = ∅) {
      从 e1 中任取一条边(u,v);
      cset = cset ∪ {u,v};
      从 e1 中删去与 u 和 v 相关联的所有边;
      }
   return cset
}
```

算法 approxVertexCover 用 cset 来存储顶点覆盖中的各顶点。初始时 cset 为空，然后在算法的循环中不断从边集 $e1$ 中选取一边 (u, v)，将边的端点加入 cset 中，并将 $e1$ 中已被 u 和 v 覆盖的边删去，直至 cset 已覆盖所有的边，即 $e1$ 为空时为止。

图 9-3 说明了算法 approx Vertex Cover 的运行情况。其中，图 9-3(a) 是作为算法输入的图 G，它有 7 个顶点和 8 条边。图 9-3(b) 表示算法选择了边 (b, c)，并将顶点 b 和 c 加入顶点覆盖 cset 中，然后将 $e1$ 中与顶点 b 和 c 相关联的边 (a, b)，(c, e)，(c, d) 和 (b, c) 从 $e1$ 中删去。图 9-3(c) 表示算法选择了边 (e, f)，并将顶点 e 和 f 加入顶点覆盖 cset 中。图 9-3(d) 表示算法

最后选择了边(d,g)。图 9-3(e) 表示算法产生的近似最优顶点覆盖 cset，它由顶点 b,c,d,e,f,g 所组成。图 9-3(f) 是图 G 的一个最小顶点覆盖，它只含有 3 个顶点：b、d 和 e。

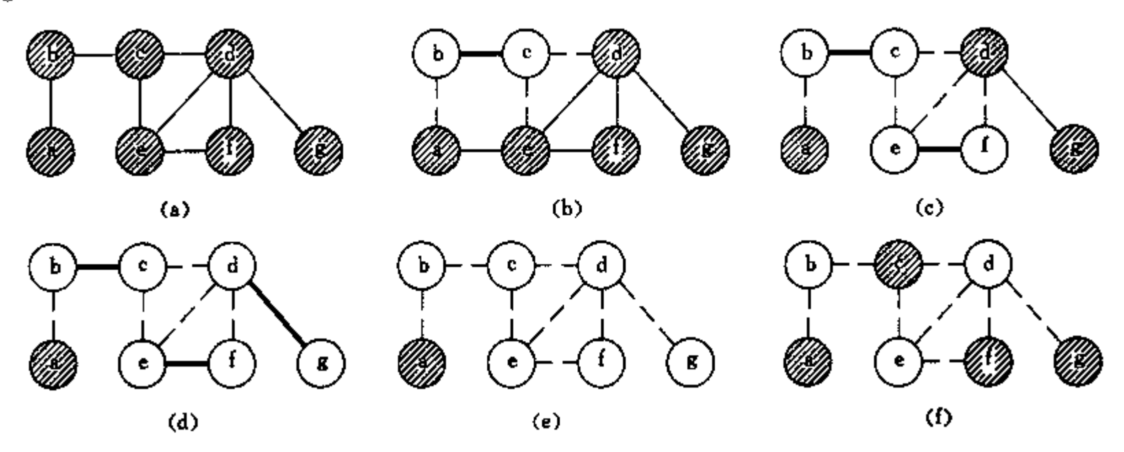


图 9-3　顶点覆盖问题的近似算法

下面考察近似算法 approx Vertex Cover 的性能。若用 A 来记算法循环中选取出的边的集合，则 A 中任何两条边没有公共端点。因为算法选择了一条边，并在将其端顶点加入顶点覆盖集 cset 后，就将 $e1$ 中与该边关联的所有边从 $e1$ 中删去。因此，下一次再选出的边就与该边没有公共端点。由数学归纳法易知，A 中各边均没有公共端点。算法终止时有 $|cset| = 2|A|$。另一方面，图 G 的任一顶点覆盖，一定包含 A 中各边的至少一个端顶点，G 的最小顶点覆盖也不例外。因此，若最小顶点覆盖为 cset*，则 $|cset^*| \geqslant |A|$。由此可得 $|cset| \leqslant 2|cset^*|$。也就是说算法 approx Vertex Cover 的性能比为 2。

9.4.3　旅行售货员问题近似算法

以最优化形式提出的旅行售货员问题可描述为：给定一个完全无向图 $G = (V, E)$，其每一边 $(u,v) \in E$ 有一非负整数费用 $c(u,v)$。我们要找出 G 的最小费用哈密顿回路。

从实际应用中抽象出的旅行售货员问题常具有一些特殊性质。比如，费用函数 c 往往具有三角不等式性质，即对任意的 3 个顶点 $u,v,w \in V$，有 $c(u,w) \leqslant c(u,v) + c(v,w)$。当图 G 中的顶点是平面上的点，任意两顶点间的费用就是这两点间的欧氏距离时，费用函数 c 就具有三角不等式性质。

可以证明，即使费用函数具有三角不等式性质，旅行售货员问题仍为 NP 完全问题。因此，不太可能找到解此问题的多项式时间算法。我们转而寻求解此问题的有效的近似算法。当费用函数 c 具有三角不等式性质时，我们可以设计出一个近似算法，其性能比为 2。而对于一般情况下的旅行售货员问题则不可能设计出具有常数性能比的近似算法，除非 $P = NP$。

1. 具有三角不等式性质的旅行售货员问题

对于给定的无向图 G，可以利用找图 G 的最小生成树的算法设计一个找近似最优的旅行售货员回路的算法。当费用函数满足三角不等式时，算法找出的旅行售货员回路的费用不会超过最优旅行售货员回路费用的 2 倍。

```
void approx TSP (Cragh g)
{
```

(1) 选择 g 的任一顶点 r;

(2) 用 Prim 算法找出带权图 g 的一棵以 r 为根的最小生成树 T;

(3) 前序遍历树 T 得到的顶点表 L;

(4) 将 r 加入到表 L 的末尾,按表 L 中顶点次序组成回路 H,作为计算结果返回;

}

图9-4 说明了算法 approxTSP 的运行情况。

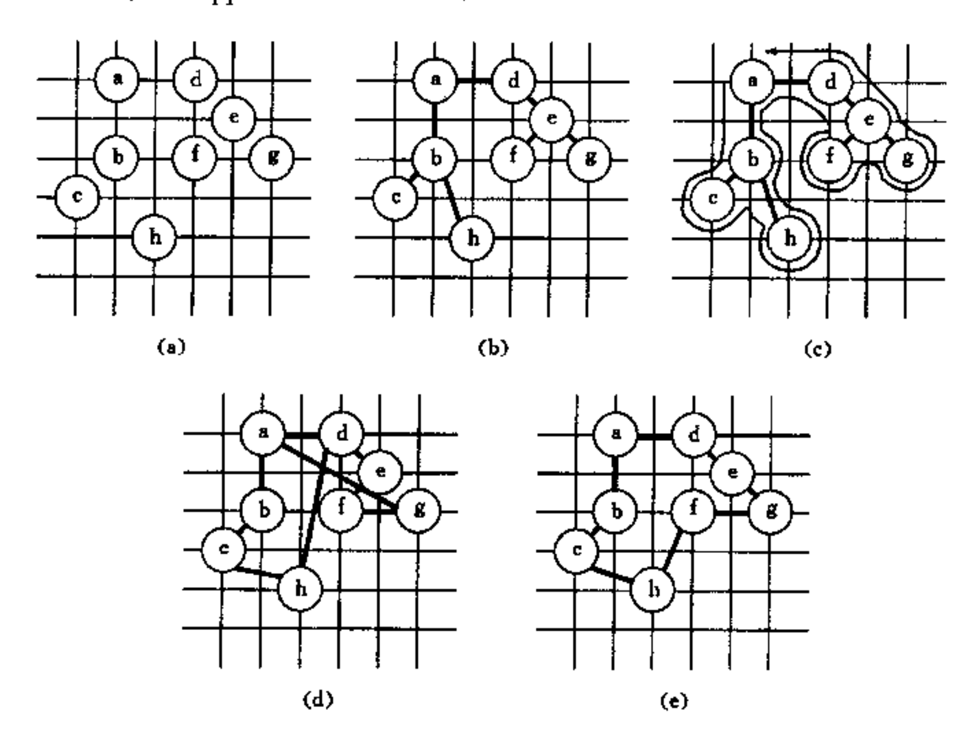


图 9-4 旅行售货员问题的近似算法

其中,图 9-4(a) 表示所给的图 G 的顶点集;图 9-4(b) 表示由算法找到的一棵最小生成树 T;图 9-4(c) 表示对树 T 所作的前序遍历访问各顶点的次序;图 9-4(d) 表示由 T 的前序遍历顶点表 L 产生的哈密顿回路 H;图 9-4(e) 是 G 的一个最小费用旅行售货员回路。

图中各顶点表示平面上的一个点。图中方格的边长为 1。各顶点间的边费用为顶点间的欧氏距离,因而费用函数满足三角不等式。从该例算出的近似最优旅行售货员回路 H 可看出,最小费用要比 H 的费用少约 23%。

由于图 G 是一个完全图,易知算法 approx TSP 的计算时间为 $\theta(|E|) = \theta(|V|^2)$。算法中没有明显地用到费用函数的三角不等式性质。因此,该算法也适用于一般的旅行售货员问题。当费用函数满足三角不等式时,该算法具有较好的性能比,即对于任何无向完全图 G,算法具有一个常数性能比 2。换句话说,若用 H^* 记图 G 的最小费用旅行售货员回路,而用 H 记算法 approx TSP 计算出的近似最优的旅行售货员回路,则 $c(H) \leqslant 2c(H^*)$。其中,$c(A) = \sum_{(u,v)\in A} c(u,v)$。下面证明这一结论。

设 T 是算法 approx TSP 计算出的图 G 的最小生成树。从 H^* 中任意删去一条边后,可得到图 G 的一棵生成树。由于 T 是最小生成树,故有 $c(T) \leqslant c(H^*)$。对树 T 所做的一个完全遍历是在访问 T 的一个顶点时列出该顶点,而在结束对 T 的一棵子树的访问并沿途返回时也列

出返回时经过的顶点。设 W 是对 T 依前序所做的完全遍历。例如,在图9-4(b)中,对 T 所做的完全遍历为 W = abcbhbadefegeda。由于对 T 所做的完全遍历 W 经过 T 的每条边恰好两次,所以有 $c(W) = 2c(T) \leqslant 2c(H^*)$。然而 W 还不是一个旅行售货员回路,它访问了图 G 中某些顶点多次。由于费用函数满足三角不等式,我们可以在 W 的基础上,从中删去已访问过的顶点,而不会增加旅行费用。若在 W 中删去顶点 u 和 w 间的一个顶点 v,就用边 (u, w) 代替原来从 u 到 w 的一条路。反复用这个办法删去 W 中多次访问的顶点可得到 G 的一条旅行售货员回路。在图9-4所示的例子中,从 W 中删去重复访问顶点后得到的回路为 H = abchdefga。这就是算法 approx TSP 计算出的近似最优哈密顿回路。由费用函数的三角不等式性质即知 $c(H) \leqslant c(W) \leqslant 2c(H^*)$。也就是说,算法 approxTSP 的性能比为2。

2. 一般的旅行售货员问题

尽管算法 approxTSP 也可用于解一般的旅行售货员问题,但我们不能保证它具有好的性能比。在费用函数不一定满足三角不等式时,不存在具有常数性能比的解 TSP 问题的多项式时间近似算法,除非 $P = NP$。换句话说,若 $P \neq NP$,则对任意常数 $\rho > 1$,不存在性能比为 ρ 的解旅行售货员问题的多项式时间近似算法。事实上,假设有一个解旅行售货员问题的近似算法 A。其性能比为 $\rho \geqslant 1$。不失一般性,可设 ρ 为一正整数,因若不然,可用 $\lceil\rho\rceil$ 来代替 ρ。在这个假设下,我们可以利用算法 A 来设计一个解哈密顿回路问题的多项式时间算法。由于哈密顿回路问题是 NP 完全的,故找到了它的一个多项式时间算法就证明了 $P = NP$。因此,在 $P \neq NP$ 的前提下,对任意 $\rho \geqslant 1$ 这样的算法 A 是不存在的。

下面说明如何用算法 A 来解哈密顿回路问题。设图 $G = (V, E)$ 是哈密顿回路问题的一个实例,要求判定 G 是否有一条哈密顿回路。为了利用算法 A 来解 G 的哈密顿回路问题,将 G 变换为旅行售货员问题的一个实例 $\langle G1, c\rangle$ 如下。其中,$G1$ 是顶点集 V 上的一个完全图,即 $G1 = (V, E1)$,$E1 = \{(u, v) \mid u, v \in V$ 且 $u \neq v\}$。$E1$ 中每一边的费用 $c(u, v)$ 定义为

$$c(u,v) = \begin{cases} 1 & (u,v) \in E \\ \rho \mid V \mid + 1 & (u,v) \in E1 - E \end{cases}$$

如上定义的图 $G1$ 和费用函数 c 显然可根据图 G 在关于 $|V|$ 和 $|E|$ 的多项式时间内构造出来。

现在考虑旅行售货员问题 $\langle G1, c\rangle$。若原图 G 有一哈密顿回路 H,则费用函数 c 赋给 H 中每边的费用均为1。因此 $\langle G1, c\rangle$ 含有一个费用为 $|V|$ 的旅行售货员回路。另一方面,若 G 中不存在哈密顿回路,则 $G1$ 的任一回路必用到了不在 E 中的边。因此,$\langle G1, c\rangle$ 的任一旅行售货员回路的费用至少为 $(\rho \mid V \mid + 1) + (\mid V \mid - 1) > \rho \mid V \mid$。

若用算法 A 来解旅行售货员问题 $\langle G1, c\rangle$,则它求出的近似最优的旅行售货员回路 H 的费用 $c(H)$ 不超过最优旅行售货员回路 H^* 的费用的 ρ 倍,即 $c(H) \leqslant \rho c(H^*)$。

当 G 有哈密顿回路 H 时,易知 $c(H) = c(H^*) = |V|$,而由算法 A 找到的旅行售货员回路 H 的费用 $c(H) \leqslant \rho c(H^*) = \rho \mid V \mid$。由上面的分析可知,$H$ 中每一条边均属于 E。故 H 也是 G 的一条哈密顿回路。

反之,若算法 A 找出的旅行售货员回路 H 的费用 $c(H) > \rho \mid V \mid$,则 $\rho \mid V \mid < c(H) \leqslant \rho c(H^*)$。由此可知 $c(H^*) > |V|$,即 $\langle G1, c\rangle$ 的最优旅行售货员回路 H^* 的费用 $c(H^*) > |V|$。由上面的分析即知,此时 G 中不存在哈密顿回路。因此,算法 A 求出 $\langle G1, c\rangle$ 的近似最优的旅行售货员回路 H 后,只要再判断一下,其费用 $c(H)$ 是否为 $|V|$,即可判定 G 是否有一条

哈密顿回路。由假设知，算法 A 可在多项式时间内完成，故可在多项式时间内解哈密顿回路问题。在 $P \neq NP$ 的前提下，这是不可能。因此，我们所假设的这样的算法 A 在 $P \neq NP$ 的前提下也是不存在的。

9.4.4 集合覆盖问题的近似算法

集合覆盖问题是一个最优化问题，其原型是多资源选择问题。集合覆盖问题可以看作是图的顶点覆盖问题的推广，它也是一个 NP 难问题。

集合覆盖问题的一个实例 $\langle X, F\rangle$ 由一个有限集 X 及 X 的一个子集族 F 组成。子集族 F 覆盖了有限集 X。也就是说，X 中每一元素至少属于 F 中的一个子集，即 $X = \bigcup_{S \in F} S$。对于 F 的一个子集 $C \subseteq F$，若 C 中的 X 的子集覆盖了 X，即 $X = \bigcup_{S \in C} S$，则称 C 覆盖了 X。集合覆盖问题就是要找出 F 中覆盖 X 的最小子集 C^*，使得

$$|C^*| = \min\{|C| \mid C \subseteq F \text{ 且 } C \text{ 覆盖 } X\}$$

图 9-5 是集合覆盖问题的一个例子。

其中，用 12 个黑点表示集合 X，$F = \{S_1, S_2, S_3, S_4, S_5, S_6,\}$。容易看出，对于这个例子，最小集合覆盖为 $C = \{S_3, S_4, S_5,\}$。

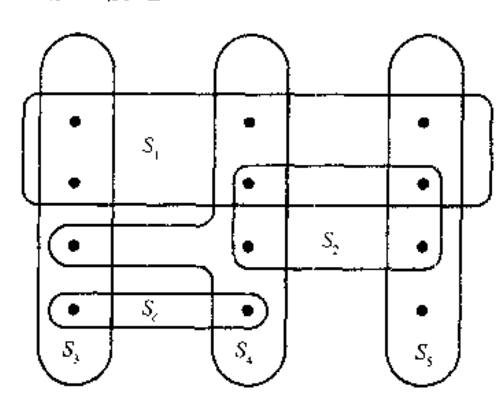


图 9-5 集合覆盖问题的一个实例 $\langle X, F\rangle$

集合覆盖问题是对许多常见的组合问题的抽象。例如，假设 X 表示解决某一问题所需的各种技巧的集合，且给定一个可用来解决该问题的人的集合，其中每个人掌握若干种技巧。我们希望从这些人的集合中选出尽可能少的人组成一个委员会，使得 X 中的每一种技巧，都可以在委员会中找到掌握该技巧的人。这个问题实质上就是一个集合覆盖问题。集合覆盖问题是一个 NP 完全问题。

对于集合覆盖问题，我们可以设计出一个简单的贪心算法，求出该问题的一个近似最优解。这个近似算法具有对数性能比，算法描述如下：

```
Set greedySetCover(X,F)
{
  U = X;
  C = ∅;
  while (U ! = ∅) {
  选择 F 中使|S∩U|最大的子集 S;
  U = U - S;
  C = C∪{S};
  }
return C;
}
```

在算法 greedySetCover 中，集合 U 用于存放在每一阶段中尚未被覆盖的 X 中元素。集合 C

包含了当前已构造的覆盖。算法的循环体是整个算法的主体。在该循环体中，首先选择 F 中覆盖了尽可能多的未被覆盖元素的子集 S。然后，将 U 中被 S 覆盖的元素删去，并将 S 加入 C。算法结束时，C 中包含了覆盖 X 的 F 的一个子集族。例如，对于图9-5 中的例子，算法 greedySetCover 依次选出子集 S_1, S_4, S_5 和 S_3 构成子集族 C。

算法 greedySetCover 的循环体最多执行 $\min\{|X|, |F|\}$ 次。而循环体内的计算显然可在 $O(|X||F|)$ 时间内完成。因此，算法的计算时间为 $O(|X||F|\min\{|X|, |F|\})$。由此即知，greedySetCover 是一个多项式时间算法。

从图9-5所给的例子可以看出，算法 greedySetCover 得到的只是集合 X 的近似最优覆盖。下面进一步考虑算法 greedySetCover 的性能比。为叙述方便，我们用 $H(d)$ 来记第 d 级调和数，即 $H(d) = \sum_{i=1}^{d} \frac{1}{i}$。可以证明，算法 greedySetCover 的性能比为 $H(\max_{S\in F}\{|S|\})$。证明过程如下。

首先对于每一个由算法 greedySetCover 选出的集合赋予其一个费用，并将这个费用分布于初次被覆盖的 X 中的元素上。然后，再利用这些费用导出所需要的算法 greedySetCover 的性能比。设 S_i 表示由算法 greedySetCover 的 while 循环所选出的第 i 个子集。在算法将 S_i 加入子集族 C 时，赋予 S_i 一个费用 1，并将这个费用平均地分摊给 S_i 中刚被覆盖的 X 中元素，即 $S_i - \bigcup_{j=1}^{i-1} S_j$ 中的元素。对每一个 $x \in X$，用 C_x 表示元素 x 摊到的费用。注意，每个元素 x 在它第一次被覆盖时得到费用 C_x，且只得到一次，以后不再得到费用。若 x 第一次被集合 S_i 覆盖，则

$$C_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$$

算法终止时，得到子集族 C，其总费用为 $|C|$。这个费用分布于 X 中的各元素上，即 $|C| = \sum_{x\in X} C_x$。由于 X 的最优覆盖 C^* 也是 X 的一个覆盖，故

$$|C| = \sum_{x\in X} C_x \leqslant \sum_{S\in C^*} \sum_{x\in S} C_x$$

稍后还将证明，对于子集族 F 中任一子集 S，有

$$\sum_{x\in S} C_x \leqslant H(|S|)$$

由此可得

$$|C| \leqslant \sum_{S\in C^*} H(|S|) \leqslant |C^*| H(\max_{S\in F}\{|S|\})$$

由此即知算法 greedySetCover 的性能比为

$$\left|\frac{C}{C^*}\right| \leqslant H(\max_{S\in F}\{|S|\})$$

以下证明 $\sum_{x\in S} C_x \leqslant H(|S|)$。对于任一 F 中的集合 $S \in F$ 以及 $i = 1,2,\cdots,|C|$，设 $u_i = |S - \bigcup_{j=1}^{i} S_j|$ 是算法选择了 $S_1, S_2, \cdots, S_i$ 后，S 中尚存的未被覆盖元素的个数。其中，u_0 定义为初始时 S 中未被覆盖的元素个数，即 $u_0 = |S|$。进一步设 k 是数列 $u_0, u_1, u_2, \cdots$ 中第一个等于 0 的下标。那么，S 中的元素被集合 $S_1, S_2, \cdots, S_k$ 所覆盖，且 $u_{i-1} \geqslant u_i$，S 中有 $u_{i-1} - u_i$ 个元素被 S_i 第一次覆盖，$i = 1,2,\cdots,k$。由此可得

$$\sum_{x\in S} C_x = \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$$

由算法的贪心选择性质可知，S 所覆盖的新元素不会比 S_i 多，否则算法将选择集合 S 而不

是 S_i。因此

$$|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}$$

由此可知

$$\sum_{x \in S} C_x \leq \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{u_{i-1}}$$

对于任意正整数 a 和 b，且 $a < b$，容易证明

$$H(b) - H(a) = \sum_{i=a+1}^{b} \frac{1}{i} \geq \frac{b-a}{b}$$

利用这个不等式我们得到

$$\begin{aligned}\sum_{x \in S} C_x &\leq \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{u_{i-1}} \leq \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\ &= H(u_0) - H(u_k) = H(u_0) - H(0) \\ &= H(u_0) = H(|S|)\end{aligned}$$

这就是我们要证明的不等式。

容易证明，对于任一正整数 n 有 $H(n) \leq \ln n + 1$。由于 $\max_{S \in F}\{|S|\} \leq |X|$，故 $H(\max_{S \in F}\{|S|\}) \leq \ln|X| + 1$。因此，也可以说，算法 greedySetCover 的性能比为 $\ln|X| + 1$。

在许多实际应用中 $\max_{S \in F}\{|S|\}$ 是一个小常数，因此由算法 greedySetCover 计算出的近似最优集合覆盖的大小只不过是最优集合覆盖的大小的一个小常数倍。例如，当一个图的顶点度数最多为 3 时，用算法 greedySetCover 解关于这个图的顶点覆盖问题，可得到一个近似最优的顶点覆盖，其性能比为 $H(3) = 11/6$。这比算法 approxVertexCover 算法得到的结果要稍好一些。

9.4.5 子集和问题的近似算法

设子集和问题的一个实例为 $\langle S, t\rangle$。其中，$S = \{x_1, x_2, \cdots, x_n\}$ 是一个正整数的集合，t 是一个正整数。子集和问题是要判定是否存在 S 的一个子集 $S1$，使得 $\sum_{x \in S1} x = t$。

该问题是一个 NP 完全问题。在实际应用中，我们常遇到的是最优化形式的子集和问题。在这种情况下，要找出 S 的一个子集 $S1$，使得其和不超过 t，但又尽可能地接近 t。例如，在第 5 章中已讨论过的最优装载问题实质上就是一个最优化形式的子集和问题。

下面先提出一个解最优化形式的子集和问题的指数时间算法，然后将这个算法作适当修改，使它成为解子集和问题的一个完全多项式时间的近似格式。

1. 解子集和问题的指数时间算法

设 L 是一个由正整数组成的表，x 是另外一个正整数。用 $L + x$ 来表示对表 L 中每个整数加上 x 后得到的新表。例如，若 $L = \langle 1,2,3,5,9\rangle$，则 $L + 2 = \langle 3,4,5,7,11\rangle$。对于整数集合 S，用记号 $S + x$ 来表示集合 S 中每个元素都加上 x，即 $S + x = \{s + x \mid s \in S\}$。

下面要描述的解子集和问题的算法 exactSubsetSum 以集合 $S = \{x_1, x_2, \cdots, x_n\}$ 和目标值 t 作为输入。算法中用到将两个有序表 $L1$ 和 $L2$ 合并成一个新的有序表的算法 mergeLists($L1$, $L2$)。与合并排序算法中用到的 Merge 算法类似，算法 mergeLists 的计算时间为 $O(|L1| + |L2|)$。

```
int exactSubsetSum (S,t)
```

```
    }
    int n = |S|;
    L[0] = {0};
    for (int i = 1;i <= n;i ++) {
      L[i] = mergeLists(L[i - 1],L[i - 1] + S[i]);
      删去 L[i] 中超过 t 的元素;
      }
    return max(L[n]);
  }
```

用 P_i 表示 $\{x_1, x_2, \cdots, x_i\}$ 的所有可能的子集和,即 P_i 中的一个元素是 $\{x_1, x_2, \cdots, x_i\}$ 的一个子集和。约定一个空集的子集和为 0,并约定 $P_0 = \{0\}$。不难用数学归纳法证明:

$$P_i = P_{i-1} \cup (P_{i-1} + x_i),\ i = 1,2,\cdots,n$$

例如,若 $S = \{1,4,5\}$,则 $P_0 = \{0\}$; $P_1 = \{0,1\}$; $P_2 = \{0,1,4,5\}$; $P_3 = \{0,1,4,5,6,9,10\}$。

由此易知,算法 exactSubsetSum 中的表 L[i] 是一个包含了 P_i 中所有不超过 t 的元素的有序表。因此,L[n] 中的最大元素 max(L[n]) 就是 S 中不超过 t 的最大子集和。

由于 P_i 中包含了所有可能的 $\{x_1, x_2, \cdots, x_i\}$ 的子集和,因此 $|P_i| = 2^i$。在最坏情况下,L[i] 可能与 P_i 相同。因此,在最坏情况下 $|L[i]| = 2^i$。由此可知,在一般情况下,算法 exactSubsetSum 是一个指数时间算法。

2. 子集和问题的完全多项式时间近似格式

基于算法 exactSubsetSum,通过对表 L[i] 作适当的修整建立一个子集和问题的完全多项式时间近似格式。在对表 L[i] 进行修整时,要用到一个修整参数 δ,$0 < \delta < 1$。用参数 δ 修整一个表 L 是指从 L 中删去尽可能多的元素,使得每一个从 L 中删去的元素 y,都有一个修整后的表 $L1$ 中的元素 z 满足 $(1 - \delta)y \leq z \leq y$。可以将 z 看作是被删去元素 y 在修整后的新表 $L1$ 中的代表。也就是说,对每一个删去元素 y,可以用新表 $L1$ 中一个元素 z 来代表 y,使得 z 相对于 y 的相对误差不超过 δ。

例如,若 $\delta = 0.1$,且 $L = \langle 10,11,12,15,20,21,22,23,24,29 \rangle$,则用 δ 对 L 进行修整后得到 $L1 = \langle 10,12,15,20,23,29 \rangle$。其中被删去的数 11 由 10 来代表,21 和 22 由 20 来代表,24 由 23 来代表。

经修整后的新表 $L1$ 中的元素也是原表 L 中的元素。对一个表进行修整后,可大大减少其中的元素个数,而对每个被删除的元素保留一个与其很接近的代表,以控制计算结果的相对误差。

下面的算法 trim 对有序表 L 进行修整,它以有序表 $L = \langle y_1, y_2, \cdots, y_m \rangle$ 作为输入,L 中元素以非减次序排列。

```
List trim(L,δ)
{
  int m = |L|;
```

```
    L1 = 〈L[1]〉;
    int last = L[1];
    for (int i = 2;i < = m;i + +) {
    if (last < (1 - δ) * L[i])
       将 L[i] 加入表 L1 的尾部;
       last = L[i];
       }
    return L1;
  }
```

在算法 trim 中，以递增的次序逐个扫描表 L 中的元素。当被扫描元素是表 L 中第一个元素或被扫描元素不能用最近加入新表 L 的元素 last 代表时，将被扫描元素加入新表 $L1$ 的尾部。而能够被 last 代表的元素不加入 $L1$，意味着该元素被删去。算法 trim 的计算时间为 $\theta(m)$。

由算法 trim 可以构造子集和问题的近似格式 approxSubsetSum 如下。该近似格式的输入参数是 n 个整数的集合 $S = \{x_1, x_2, \cdots, x_n\}$、目标整数 t 和一个近似参数 ε，$0 < \varepsilon < 1$。

```
int approxSubsetSum(S,t,ε)
{
  n = |S|;
  L[0] = 〈0〉;
  for (int i = 1;i < = n;i + +) {
    L[i] = mergeLists(L[i - 1],L[i - 1] + S[i]);
    L[i] = trim(L[i],ε/n);
     删去 L[i] 中超过 t 的元素;
    }
  return max(L[n]);
}
```

在上述算法中，首先将 $L[0]$ 初始化为只含一个 0 元素的表。然后在算法的主循环中逐次计算表 L[i]，$i = 1,2,\cdots,n$。计算出的表 L[i] 实际上就是对集合 P_i 进行修整后的有序表，修整参数为 $\delta = \varepsilon/n$。另外，$L[i]$ 中已将超过目标整数 t 的元素及时删除，以减少不必要的计算。

我们用一个例子来说明 approxSubsetSum 的运行情况。在该例中，S = 〈104,102,201,101〉，$t = 308$，$\varepsilon = 0.2$。由算法确定的修整参数 δ 是 $\varepsilon/4 = 0.05$。初始时，L[0] = 〈0〉。在算法的主循环中逐次计算出 L[1]，L[2]，L[3] 和 L[4]。每次计算经过合并、修整和删除大于 t 的元素 3 个阶段。现将算法计算 L[i]，$i = 1,2,3,4$ 的 3 个阶段的计算结果列出如下：

L[1] = 〈0,104〉;

L[1] = 〈0,104〉;

L[1] = 〈0,104〉;

L[2] = 〈0,102,104,206〉;

L[2] = 〈0,102,206〉;

$L[2] = \langle 0,102,206 \rangle$;

$L[3] = \langle 0,102,201,206,303,407 \rangle$;

$L[3] = \langle 0,102,201,303,407 \rangle$;

$L[3] = \langle 0,102,201,303 \rangle$;

$L[4] = \langle 0,101,102,201,203,302,303,404 \rangle$;

$L[4] = \langle 0,101,201,302,404 \rangle$;

$L[4] = \langle 0,101,201,302 \rangle$;

算法最后返回 $z = 302$ 作为近似解答。容易看出,该例的最优解为 $104 + 102 + 101 = 307$。近似解的相对误差在 2% 以内。在理论上,算法可以保证对子集和问题的任一实例可用,其相对误差在 ε 之内。

下面进一步讨论算法 approxSubsetSum 的性能。通过分析可以得出如下结论:

(1) 算法 approxSubsetSum 计算出的近似解是 S 的一个子集和,它关于最优解的相对误差不超过预先给定的误差界 ε。

(2) 算法 approxSubsetSum 是子集和问题的一个完全多项式时间近似格式,即它的计算时间是关于输入规模 n 和 $1/\varepsilon$ 的多项式。

首先注意到,算法中对 $L[i]$ 进行修整,并将其中超过 t 的元素删去后,$L[i]$ 中每个元素仍为集合 P_i 的成员。因此,算法返回的 z 值是 P_n 的成员,从而它是 S 的一个子集和。若设子集和问题的最优值为 C^*,则算法返回的近似最优值 z 与 C^* 的相对误差为 $\frac{C^* - z}{C^*} = 1 - \frac{z}{C^*}$。我们要证明这个相对误差不超过 ε,即 $1 - z/C^* \leqslant \varepsilon$。这等价于 $z \geqslant (1-\varepsilon)C^*$。注意到在对 $L[i]$ 进行修整时,被删除元素与其代表元素的相对误差不超过 ε/n。对修整次数 i 用数学归纳法容易证明,对于 P_i 中任一不超过 t 的元素 y,有 $L[i]$ 中一个元素 x,使得 $(1-\varepsilon/n)^i y \leqslant x \leqslant y$。由于最优值 $C^* \in P_n$,故存在 $x \in L[n]$,使得 $(1-\varepsilon/n)^n C^* \leqslant x \leqslant C^*$。又因为算法返回的是 $L[n]$ 中最大元素 z,故有 $x \leqslant z \leqslant C^*$。因此,$(1-\varepsilon/n)^n C^* \leqslant z \leqslant C^*$。最后,由于 $(1-\varepsilon/n)^n$ 是 n 的递增函数,因此,当 $n > 1$ 时,有 $(1-\varepsilon) \leqslant (1-\varepsilon/n)^n$。由此可得,$(1-\varepsilon)C^* \leqslant z \leqslant C^*$。这就证明了算法 approxSubsetSum 返回的近似最优值 z 关于最优值 C^* 的相对误差不超过 ε。

从算法 approxSubsetSum 的循环体可以看出,每次对有序表 $L[i]$ 所作的合并、修整和删除超过 t 的元素的计算时间为 $O(|L[i]|)$。因此,整个算法的计算时间不会超过 $O(n|L[n]|)$。注意到算法对表 $L[i]$ 进行修整后,表中相继元素 a 和 b 间满足 $a/b > 1/(1-\varepsilon/n)$。也就是说,表 $L[i]$ 相继元素间至少相差一个比例因子 $1/(1-\varepsilon/n)$。而表 $L[i]$ 中最大数不会超过 t。因此,算法完成了对 $L[i]$ 的合并、修整和删除超过 t 的元素等操作后,$L[i]$ 中元素个数不超过

$$\frac{\ln t}{\ln(1/(1-\varepsilon/n))} = \frac{\ln t}{-\ln(1-\varepsilon/n)} \leqslant \frac{t}{\varepsilon/n} = \frac{nt}{\varepsilon}$$

特别地,$|L[n]| \leqslant \frac{nt}{\varepsilon}$。于是,算法 approxSubsetSum 的计算时间为 $O(n^2/\epsilon)$。这表明它是一个完全多项式时间近似格式。

习题 9

9-1 试写出完成下面计算的 RAM 和 RASP 程序:

(1) 给定输入 n，计算 $n!$。

(2) 读入 n 个正整数(用 0 做结束标志)，然后，按从大到小的顺序输出这 n 个数。

9-2　用对数耗费和均匀耗费两种标准分析习题9-1 中程序的时间和空间复杂性

9-3　写出一个计算 n^n 的 RAM 程序，要求该程序在均匀耗费标准下的时间复杂性为 $O(\log n)$，并证明程序的正确性。

9-4　设问题 P 关于实例 I 的精确解为 $c^*(I)$，解问题 P 的近似算法 A 对于实例 I 得到的近似解为 $c(I)$。如果存在一常数 k，使得对于 P 的任何实例 I 均有

$$|c^*(I) - c(I)| \leqslant k$$

则称算法 A 是解问题 P 的绝对近似格式。

平面图的色数问题是对于给定的平面图 $G = (V, E)$，确定对其顶点着色的最小色数。试设计解平面图着色问题的一个多项式时间绝对近似算法 A 使得 $|c^*(I) - c(I)| \leqslant 1$。

9-5　设有 n 个程序 $1,2,\cdots,n$ 要存入两张容量为 M 的磁盘中。第 i 个程序需要的存储空间为 m_i，$i = 1,2,\cdots,n$。设计一个算法计算出这两张磁盘能存放的最多程序个数。

(1) 证明上述问题是 NP 难的

(2) 下面的算法 pStore 是解上述问题的一个绝对近似算法：

```
int pStore(int n, int maxM, int * m)
{
    sort(m,n); // 将 m 从小到大排序
    int i = 1;
    for (int j = 1;j <= 2;j++) {
      int sum = 0;
      while (sum + m[i] <= maxM) {
       System.out.println("sort program" + i + "on disk" + j);
       sum += m[i];
       if (i == n) return i;
       else i++;
      }
    }
    return i - 1;
}
```

试证明对于上述算法 pStore 有 $|c^*(I) - c(I)| \leqslant 1$。

9-6　设计一个有效的贪心算法，使其能在线性时间内找到一棵树的最优顶点覆盖。

9-7　解顶点覆盖问题的一个启发式算法如下：每次选择具有最高度数的顶点，然后将与其关联的所有边删去。举例说明该算法的性能比将大于 2。

9-8　一个图 G 的最优顶点覆盖是其补图中最大团集的补集。这个关系是否暗示对于团问题也有一个常数性能比的近似算法？

9-9　试设计解 TSP 问题的 $O(n^3)$ 时间近似算法，使其性能比达到 1.5。

9-10　证明旅行售货员问题的一个实例可在多项式时间内变换为该问题的另一个实例，

使得其费用函数满足三角不等式，且两实例具有相同的最优解。说明是否可以通过这个变换使得一般的旅行售货员问题具有一个常数性能比的近似算法。

9-11 瓶颈旅行售货员问题是要找出图 G 的一条哈密顿回路，且使回路中最长边的长度最小。若费用函数满足三角不等式，给出解此问题的性能比为 3 的近似算法（提示：递归地证明，可以通过对 G 的最小生成树进行完全遍历并跳过某些顶点，但不能跳过多于两个连续的中间顶点，以此方式来访问最小生成树中每个顶点恰好一次）。

9-12 若旅行售货员问题中，图 G 的各顶点均为平面上的点，且费用函数 $c(u,v)$ 定义为点 u 和 v 之间的欧氏距离，证明 G 的一个最优旅行售货员回路不会自相交。

9-13 试说明如何实现算法 greedySetCover，使其计算时间为 $O\left(\sum_{S\in F}|S|\right)$。

9-14 试给出一族集合覆盖问题的实例，用以说明算法 greedySetCover 可以产生的不同解的个数随实例规模指数增长。这里所说的不同解是指算法 greedySetCover 在作贪心选择时可以有多种选择，即使 $|S\cap U|$ 最大的子集可有多个时，不同的选择导致算法的一个不同的解。

9-15 如何修改近似算法 approxSubsetSum，使其可以找出子集和不小于 t 的最小子集和？

9-16 多机调度问题。设有 m 台完全相同的机器来完成 n 个彼此独立的任务，第 i 个任务所需的机器时间为 t_i，$i=1,2,\cdots,n$。我们要确定一个时间表，使全部 n 个任务都结束的时间最短。

解上述问题的最长处理时间算法 LPT 每次从待安排任务中选择最长处理时间的任务，并安排给一台完全空闲机器。试在 $O(n\log n)$ 时间内实现算法 LPT，并证明该算法所得到的解的相对误差

$$\lambda=\left|\frac{c^*-c}{c^*}\right|\leqslant\frac{1}{3}-\frac{1}{3m}$$

9-17 LPT 算法的最坏情况实例。设 $n=2m+1$ 且 $t_i=2m-\lfloor(i+1)/2\rfloor$，$1\leqslant i\leqslant 2m$，$t_{2m+1}=m$。试构造多机调度问题关于该实例的最优解 c^* 和用算法 LPT 求出的解 c，并计算近似算法 LPT 的性能比

$$\eta=\left|\frac{c^*-c}{c^*}\right|$$

9-18 设在多机调度问题中，要在所给 m 台机器上安排的 n 个任务已按各自所需处理时间的递减序列排列 $t_1\geqslant t_2\geqslant\cdots\geqslant t_n$。解此问题的算法 LPT2 先确定一个正整数 k，对前 k 个任务求最优安排，然后对后 $n-k$ 个任务用算法 LPT（习题9-16）求解。

（1）试证明算法 LPT2 的解的相对误差

$$\lambda\leqslant\frac{1-1/m}{1+\lfloor k/m\rfloor}$$

（2）根据（1）的结论，设计一个解多机调度问题的多项式时间近似算法，对于给定的 $\varepsilon>0$，算法所需的计算时间为 $O(n\log n+m^{m/\varepsilon})$。

9-19 设 α 是一个含有 n 个变量和 m 个合取项的合取范式。关于 α 的最大可满足性问题要求确定 α 的最多个数的合取式，并使这些合取式可同时满足。设 k 是 α 的所有合取式中因子个数的最小值。证明下面的解最大可满足问题的近似算法 mSAT 的相对误差为 $\frac{1}{k+1}$。

```
Set mSAT(a)
{// x_i,1 ≤ i ≤ n,是 a 中 n 个变量;C_i,1 ≤ i ≤ m,是 a 的 m 个合取项
    cl = ∅;
    left = {C_i | 1 ≤ i ≤ m};
    lit = {x_i, x̄_i | 1 ≤ i ≤ n};
    while (lit 含有在 left 的合取式中出现的因子) {
        设 y 是 lit 的在 left 的合取式中出现次数最多的因子;
        设 r 是 left 中含有因子 y 的所有合取式的集合;
        cl = cl ∪ r;
        left = left - r;
        lit = lit - {y, ȳ};
        }
    return (cl);
}
```

9-20　试证明下面的解最大可满足问题的近似算法 mSAT2 的相对误差为$\frac{1}{2^k}$，k 是 a 的所有合取式中因子个数的最小值。

```
Set mSAT2(a)
{//x_i,1 ≤ i ≤ n,是 a 中 n 个变量;c_i,1 ≤ i ≤ m,是 a 的 m 个合取项
    for (int i = 1;i <= m;i++) w[i] = 2^{-|c_i|};
    cl = ∅;
    left = {c_i | 1 ≤ i ≤ m};
    lit = {x_i, x̄_i | 1 ≤ i ≤ n};
    while (lit 含有在 left 的合取式中出现的因子) {
        设 y 是 lit 的在 left 的合取式中出现的因子;
        设 r 是 left 中含有因子 y 的所有合取式的集合;
        设 s 是 left 中含有因子 ȳ 的所有合取式的集合;
        if( Σ_{c_i∈r} w[i] ≥ Σ_{c_i∈s} w[i]){
          cl = cl ∪ r;
          left = left - r;
          对所有 c_i ∈ s, w[i] = 2 * w[i];
          }
        else {
          cl = cl ∪ s;
          left = left - s;
          对所有 c_i ∈ r, w[i] = 2 * w[i];
          }
```

```
            lit = lit - {y,y};
            }
        return (cl);
    }
```

附录　C++概要

本附录将简要介绍C++语言的基本知识。

1. 变量、指针和引用

(1) 变量

变量是程序设计语言对存储单元的抽象，它具有以下属性：

变量名(name)　变量名是用于标识变量的符号。

地址(address)　地址是变量所占据的存储单元的地址。变量的地址属性也称为左值。

大小(size)　变量的大小指该变量所占据的存储空间的数量(以字节数来衡量)。

类型(type)　变量的类型指变量所取的值域以及对变量所能执行的运算集。

值(value)　变量的值是指变量所占据的存储单元中的内容。这些内容的意义由变量的类型所决定。变量的值属性也称为右值。

生命期(lifetime)　变量的生命期是指在执行程序期间变量存在的时段。

作用域(scope)　变量的作用域是指在程序中变量被引用的语句范围。

(2) 指针变量

C++中的指针变量是一个Type*类型的变量。其中Type为任一已定义的类型。指针变量用于存放对象的存储地址。例如：

```
int n=8;
int *p;
p=&n;
int k=*p;
```

其中，p是一个指向int类型的指针。通过间接引用指针来存取指针所指向的变量。

(3) 引用

在C++中，引用是变量的一个替代名。引用的定义与变量的定义很相似，但引用不是变量。

Type & 表示对一个类型为Type的变量的引用。例如：

```
int i=5;
int &j=i;
i=7;
cout<<i<<endl;
cout<<j<<endl;
```

其中，j是对变量i的一个引用。当i的值改变时，j的值也跟着改变。因此，上面的输出语句输

出的 i 和 j 的值都是 7。

2. 函数与参数传递

(1) 函数

C++中有两种函数:常规函数和成员函数。不论哪种函数,其定义都包括 4 个部分:函数名、形式参数表、返回类型和函数体。函数的使用者通过函数名来调用该函数。调用函数时,将实际参数传递给形式参数作为函数的输入。函数体中的处理程序实现该函数的功能。最后将得到的结果作为返回值输出。例如,下面的函数 max 是一个简单函数的例子。

```
int max(int x,int y)
{   return x>y ? x:y;
}
```

其中,max 是函数名;函数名后圆括号中的 int x 和 int y 是形式参数;函数名前面的 int 是返回类型;花括号内是函数体,它实现函数的具体功能。

C++中函数一般都有一个返回值。函数的返回值表示函数的计算结果或函数执行状态。如果所定义的函数不需要返回值,可使用 void 来表示它的返回类型。函数的返回值通过函数体中的 return 语句返回。return 语句的作用是返回一个与返回类型相同类型的值,并中止函数的执行。

(2) 参数传递

在 C++中调用函数时传递给形参表的实参必须与形参在类型、个数、顺序上保持一致。参数传递有两种方式。一种是按值传递方式。在这种参数传递方式下,把实参的值传递给函数局部工作区相应的副本中。函数使用副本执行必要的计算。因此函数实际修改的是副本的值,实参的值不变。

参数传递的另一种方式是按引用传递参数。在这种参数传递方式下,需将形参声明为引用类型,即在参数名前加上符号"&"。当一个实参与一个引用类型结合时,被传递的不是实参的值,而是实参的地址。函数通过地址存取被引用的实参。执行函数调用后,实参的值将发生改变。例如:

```
void Swap(int &x,int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

函数调用 Swap(x,y)交换变量 x 和 y 的值。

在 C++中数组参数的传递属特殊情形。数组作为形参可按值传递方式声明,但事实上采用引用方式传递。实际传递的是数组第一个元素的地址。因此在函数体内对于形参数组所作的任何改变都会在实参数组中反映出来。

若传递给函数的实参是一个对象(作为类的实例),在函数中就创建了该对象的一个副本。

在创建这个副本时不调用该对象的构造函数,但在函数调用结束前要调用该副本的析构函数撤销这个副本。若采用引用方式传递对象,在函数中不创建该对象的副本,因而也不需撤销副本。但函数将改变引用传递的对象。

3. C++的类

C++的类(class)体现了抽象数据类型(ADT)的思想,它将说明与实现分离。

C++的类由4个部分组成:

(1) 类名;

(2) 数据成员;

(3) 函数成员(也称成员函数);

(4) 访问级别。

对类成员的访问有3种不同的级别:公有(public)、私有(private)和保护(protected)级别。在public域中声明的数据成员和函数成员可以在程序的任何部分访问;在private和protected域中声明的数据成员和函数成员构成类的私有部分,只能由该类的对象和成员函数,以及被声明为友员(friend)的函数或类的对象对它们进行访问。此外,在protected域中声明的数据成员和函数成员还允许该类的子类访问它们。下面是C++中定义的矩形类Rectangle的例子。

```
class Rectangle {
public:
   Rectangle(int,int,int,int);      // 构造函数
   ~Rectangle();                    // 析构函数
   int GetHeight();                 // 矩形的高
   int GetWidth();                  // 矩形的宽
private:
   int x1,y1,h,w;
   // (x1,y1)是矩形左下角点的坐标;
   // h是矩形的高;w是矩形的宽。
};

Rectangle::GetHeight() {return h;} // 返回矩形的高
Rectangle::GetWidth() {return w;} // 返回矩形的宽
```

4. 类的对象

下面的代码段说明了如何声明类Rectangle的对象,以及如何调用其成员函数。

```
Rectangle r(0,0,2,3);
Rectangle s(0,0,3,4);
Rectangle *t = &s;
if (r.GetHeight()*r.GetWidth() > t->GetHeight()*t->GetWidth())
```

```
    cout< < "矩形 r";
  else cout< < "矩形 s";
  cout< < "的面积较大。" < <endl;
```

类对象的声明与创建方式类似于变量的声明与创建方式。对一个对象成员进行访问或调用可采用直接选择(·)或间接选择(- >)来实现。

5. 构造函数与析构函数

C + +类的构造函数(constructor)用于初始化一个对象的数据成员。构造函数名与它所在的类名相同。构造函数必须声明为类的公有成员函数。构造函数不可有返回值也不得指明返回类型。例如,类 Rectangle 的构造函数可定义如下:

```
Rectangle::Rectangle(int x = 0,int y = 0,int height = 0,int width = 0)
:x1 (x), y1 (y), h (height), w (width)
{}
```

可用如下方式声明 Rectangle 的对象 r,s 和 t:

```
Rectangle r(0,0,2,3);
Rectangle  * s = new Rectangle(0,0,3,4);
Rectangle t;
```

析构函数(destructor)用于在一个对象被撤销时删除其数据成员。析构函数名也与它的类名相同,并在前面加上符号"~"。

6. 运算符重载

C + +允许为用户定义的数据类型重载运算符。

下面的代码段实现对类 Rectangle 的运算符"= ="的重载。

```
bool Rectangle::operator= =(const Rectangle &s)
{
    if (this= = &s) return true;
    if ((x1 = =s.x1)&&(y1 = =s.y1)&&(h= =s.h)&&(w = =s.w)) return true;
    else return false;
}
```

其中,用到 C + +中的保留字 this。在类的成员函数内部,this 表示一个指向调用该成员函数的对象的指针,因此该对象也可用 * this 来表示。

经重载运算符"= ="后,即可用运算符"= ="来判定两个 Rectangle 对象是否相同。

7. 友元函数

在类的声明中可使用保留字 friend 来定义友元函数。友元函数实际上并不是这个类的成

员函数。它可以是一个常规函数,也可以是另一个类的成员函数。如果想通过这个函数来存取类的私有成员和保护成员,就必须在类的声明中给出该函数的原型,并在前面加上 friend。

8. 内联函数

在函数定义前加上一个 inline 前缀,该函数就被定义成一个内联函数。内联函数的保留字 inline 告诉编译器在任何调用该内联函数的地方直接插入内联函数的函数体

9. 结构

在 C++ 中,结构(struct)与类的区别是,在结构中默认的访问级别是 public,而在类中默认的访问级别是 private。除此之外,struct 与 class 是等价的。

10. 联合

联合(union)是一种结构。在 C++ 中,联合可以包含变量和函数,还可以包含构造函数与析构函数。因此,可以用联合来定义类。C++ 的联合保留了所有 C 的特性,其中最重要的是让所有数据成员共享相同的存储地址。与结构类似,联合的默认访问级别是 public。

在使用 C++ 的联合时应注意,联合不能继承其他任何类型的类;联合不能是基类,不能包含虚成员函数;联合不能含有静态变量;如果一个对象有构造函数与析构函数,那么它不能成为联合的成员;如果一个对象重载了运算符"=",它也不能成为联合的成员。

11. 异常

C++ 的异常(exception)提供了一种处理错误的简捷方法。当程序发现一个错误,就引发一个异常,以便在程序最合适的地方捕获异常并进行处理。在 C++ 中,异常是一个对象,它是从基类 exception 派生出来的。程序通过 throw 来引发异常。例如:

```
class error {};
void f(void)
{
   //..............
   throw error();
}
```

throw 语句类似于 return 语句,但它描述函数的异常终止。异常处理程序通常用一个 try 块来定义。在引发异常之前,程序一直执行 try 块体。在 try 块体之后有一个或多个异常处理程序。每一个异常处理程序由一个 catch 语句组成。这个语句指明欲捕获的异常以及出现该异常时要执行的代码块。当 try 引发了一个已定义的异常时,控制就转移到相应的异常处理程序中。

```
void g(void)
{
   try {
     f();
```

```
    }
  catch (error) {
    异常处理程序;
    }
  catch (error1) {
    异常处理程序;
    }
}
```

12. 模板

模板(template)是 C++ 提供的一种新机制,用于增强类和函数的可重用性。

在前面讨论的函数 max 中,有两个 int 类型的参数 a 和 b。函数 max 返回 a、b 二者中较大者。如果还要求两个 double 类型对象中的较大者,就需要重新定义函数 max。通过使用模板,可以定义一个通用的函数 max 如下:

```
template <class Type>
Type max(Type x, Type y)
{
  return x > y ? x:y;
}
```

上述模板定义了一个 max 函数的家族系列,它们分别对应于不同的类型 Type。编译器根据需要创建适当的 max 函数。例如,下面的语句

```
int i = max(1,2);
double x = max(1.0,2.0);
```

将创建两个 max 函数。其中之一的参数类型为 int,另一个的参数类型为 double。

除了定义通用函数外,模板还可用于定义通用类。例如:

```
class Stack
{
public:
  void Push(int x);
  int *Pop(int& x);
private:
  int top;
  int *stack;
  int MaxSize;
};
```

这个 Stack 类描述了一个整数栈。其成员函数 Push 和 Pop 分别用于从栈中插入和删除一个 int 类型的对象。如果要求使用不同元素类型的栈，就必须写一个不同的 Stack 类。通过使用模板，我们可以定义一个通用的栈类如下：

```
template <class Type>
class Stack
{
public:
  Stack(int MaxStackSize=100);
  bool IsFull();
  bool IsEmpty();
  void Push(const Type& x);
  Type * Pop(Type& x);
private:
  int top;
  Type *stack;
  int MaxSize;
};
template <class Type>
Stack<Type>::Stack(int MaxStackSize):MaxSize(MaxStackSize)
{
   stack=new Type[MaxSize];
   top=-1;
}
template <class Type>
inline bool Stack<Type>::IsFull()
{
   if (top==MaxSize-1) return true;
   else return false;

template <class Type>
inline bool Stack<Type>::IsEmpty()
{
   if (top==-1) return true;
   else return false;
}
template <class Type>
void Stack<Type>::Push(const Type& x)
```

```
    if (IsFull()) StackFull();
    else stack[++top] = x;
}
template <class Type>
Type* Stack<Type>::Pop(Type& x)
{
    if (IsEmpty()) {StackEmpty(); return 0;}
    x = stack[top--];
    return &x;
}
```

下面的语句创建两个栈类 Stack < int > 和 Stack < double > :

```
Stack<int> S(1000);
Stack<double> T(1000);
```

13. 动态存储分配

(1) 运算符 new

C++的运算符 new 可用于动态存储分配。该运算符返回一个指向所分配空间的指针。例如,要为一个整数动态分配存储空间,可以用下面的语句说明一个整型指针变量 int *x;当需要使用该整数时,用下面的语句为它分配存储空间

```
y = new int;
```

为了在刚分配的空间中存储一个整数值 10,用下面的语句实现

```
*y = 10;
```

上述各语句的 3 种等价表达方式如下:

```
int *y = new int;
*y = 10;
或 int *y = new int(10);
或 int *y;
y = new int(10);
```

(2) 一维数组

为了在运行时创建一个大小可动态变化的一维浮点数组 x,可先将 x 声明为一个 float 类型的指针。然后用 new 为数组动态地分配存储空间。例如

```
float *x = new float[n];
```

将创建一个大小为 n 的一维浮点数组。运算符 new 分配 n 个浮点数所需的空间,并返回指向第一个浮点数的指针。然后可用 x[0],x[1],…,x[$n-1$]来访问每个数组元素。

(3) 运算符 delete

当动态分配的存储空间已不再需要时应及时释放所占用的空间。在 C++中,用运算符

delete 来释放由 new 分配的空间。例如

```
delete y;
delete []x;
```

分别释放分配给 * y 的空间和分配给一维数组 x 的空间。

(4) 二维数组

C++提供了多种声明二维数组的机制。在许多情况下,当形式参数是一个二维数组时,必须指定其第二维的大小。例如,a[][10]是一个合法的形式参数,而 a[][]则不是。为了克服这种限制,可以使用动态分配的二维数组。例如,下面的代码创建一个类型为 Type 的动态工作数组,这个数组有 rows 行和 cols 列。

```
template <class Type>
void Make2DArray(Type * * &x,int rows, int cols)
{
    x = new Type * [rows];
    for (int i = 0;i < rows;i++)
        x[i] = new Type[cols];
}
```

当不再需要一个动态分配的二维数组时,可按以下步骤释放它所占用的空间。首先释放在 for 循环中为每一行所分配的空间。然后释放为行指针分配的空间。具体实现可描述如下:

```
template <class Type>
void Delete2DArray(Type * * &x,int rows)
{
    for (int i = 0;i < rows;i++)
        delete []x[i];
    delete []x;
    x = 0;
}
```

注意在释放空间后将 x 置为 0,以防止用户继续访问已被释放的空间。

参 考 文 献

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. Data Structures and Algorithms. Addison-Wesley, 1983

2 Sara Baase. Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley, 2001

3 Michael Ben-Or. Lower Bounds for Algebraic Computation Trees. In Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, 1983. 80 ~ 86

4 J. A. Bondy and U. S. R. Murty. Graph Theory with Applications. American Elsevier, 1976

5 Gilles Brassard and Paul Bratley. Algorithmics: Theory and Practice. Prentice-Hall, 1988

6 T. H. Cormen, C. E. Leiserson, R. L. Rivest and c. stein. Introduction to Algorithms. The MIT Press, second edition, New York, McGraw-Hill, 2001

7 Herbert Edelsbrunner. Algorithms in Combinatorial Geometry. Volume 10 of EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1987

8 Jack Edmonds. Matroids and the Greedy Algorithm. Mathematical Programming, 1971, 1: 126 ~ 136

9 Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York. NY, 1979

10 G. H. Gonnet. Handbook of Algorithms and Data Structures. Addison-Wesley, 1984

11 R. L. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. Information Processing Letters, 1972, 1: 132 ~ 133

12 John E. Hopcroft and Robert E. Tarjan. Efficient Algorithms for Graph Manipulation. Communications of the ACM, 1973, 16(6): 372 ~ 378

13 E. Horowitz, S. Sahni and D. Mehta. Fundamentals of Data Structures in C++. W. H. Freeman, New York, NY, 1994

14 E. Horowitz, S. Sahni and S. Rajasekeran. Computer Algorithms/C++. Computer Science Press, 1996

15 Donald E. Knuth. Sorting and Searching. Volume 3 of The Art of Computer Programming. Addison-Wesley, 1973

16 Christos H. Papadimitriou and Kenneth Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, 1982

17 Robert E. Tarjan. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, 1983

18 K. Mehlhorn, St. Naher. LEDA A Platform of Combinatorial and Geometric Computing. Cambridge University Press, 1999